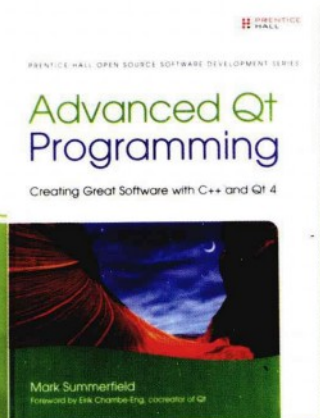


PEARSON

Qt

高级编程

Advanced Qt Programming
Creating Great Software with C++ and Qt 4



[英] Mark Summerfield 著

白建平 王军锋 闫锋欣 白净 译
高波 审校



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

<http://www.phei.com.cn>

Qt高级编程

Advanced Qt Programming
Creating Great Software with C++ and Qt 4

掌握Qt强大的API、模式和开发实践经验

Qt已经为跨平台桌面、Web和移动产品开发出一套令人瞩目的强大解决方案。然而，即使身经百战的Qt程序员也仅可用到Qt很少的一部分功能。而且，有关Qt的最新特性的实用信息正在变得越来越少——直到今天也是如此。

本书正是为向开发人员说明如何充分利用Qt 4.6最有价值的各种新API、应用程序模式和开发实践经验。本书由世界知名的Qt专家Mark Summerfield执笔，重点关注那些以最小复杂度却可提供最强大、最灵活的技术。

Summerfield特别对模型/视图、图形/视图、桌面/Web混合应用、多线程、多媒体和富文本的应用程序编程等方面的内容进行了重点论述。自始至终，作者都用真实、可下载的示例程序为Qt 4.6和未来的Qt版本进行了彻底测试。

本书特点

- 贯穿始终用带WebKit的Qt生成健壮的、多彩的桌面/Internet应用程序
- 说明如何使用Phonon框架构建强大的多媒体应用程序而无须管理底层细节
- 介绍了使用模型/视图表和树模型、委托和视图工作中涉及的各种先进技术
- 说明如何用QtConcurrent和QThread写出更为有效的多线程程序
- 包括创建富文本编辑器和文档的具体细节
- 介绍了Qt强大的动画和状态机框架

本书所含示例程序的全部源代码均可从以下站点免费下载：

www.qtrac.eu (英文)

www.qtcn.org/advqt/ (中文)



ISBN 978-7-121-13110-3



9 787121 131103 >

定价：59.00 元



策划编辑：冯小贝
责任编辑：李秦华
责任美编：孙焱津



欢迎登录 免费下载优质教学资源
<http://www.hxedu.com.cn>



本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

Qt 高级编程

Advanced Qt Programming
Creating Great Software with C++ and Qt 4

[英] Mark Summerfield 著

白建平 王军锋 闫锋欣 白 净 译
吴 迪 戚 彬 高 波 审校

电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING



内 容 简 介

本书是一本阐述 Qt 高级编程技术的书籍。本书以工程实践为主旨,是对 Qt 现有的 700 多个类和上百万字参考文档中部分关键技术深入、全面的讲解和探讨,如丰富的网络/桌面应用程序、多线程、富文本处理、图形/视图架构、模型/视图架构等;另外,除对每章主题内容的探讨外,还给出了许多与之相关的类、方法和技术细节,从而尽可能多地展示了 Qt 的各种特色。因此,即使是很有经验的 Qt 程序开发人员,也可以从书中找出自己不曾注意到的技术点。书中的全部示例程序都已用 Qt 4.6 或者 Qt 4.5 在 Windows、Mac OS X 和 Linux 系统上进行了测试。

本书主要面向 C++/Qt 程序开发人员,也适合对 Qt 编程感兴趣人员和广大的计算机编程爱好者阅读,也可作为相关机构的培训教材。

Authorized translation from the English language edition, entitled Advanced Qt Programming: Creating Great Software with C++ and Qt 4, 978-0-321-63590-7 by Mark Summerfield published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2011 Qttrac Ltd.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright © 2011.

本书简体中文版由培生教育出版亚洲有限公司授予电子工业出版社出版。专有版权受法律保护。本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签,无标签者不得销售。

版权贸易合同登记号 图字:01-2011-1800

图书在版编目(CIP)数据

Qt 高级编程/(英)萨默菲尔德(Summerfield, M.)著;白建平等译. —北京:电子工业出版社, 2011.4
书名原文: Advanced Qt Programming: Creating Great Software with C++ and Qt 4
ISBN 978-7-121-13110-3

I. ①Q… II. ①萨…②白… III. ①软件工具—程序设计 IV. ①TP311.56

中国版本图书馆 CIP 数据核字(2011)第 045128 号

策划编辑:冯小贝

责任编辑:李秦华

印 刷:涿州市京南印刷厂

装 订:涿州市桃园装订有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本:787×1092 1/16 印张:21.25 字数:544 千字

印 次:2011 年 4 月第 1 次印刷

定 价:59.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888。

质量投诉请发邮件至 zlts@phei.com.cn,盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010)88258888。

译者序

Qt 是跨平台的应用程序和用户接口 (UI) 开发框架,由集成开发工具、跨平台类库和集成开发环境 (IDE) 组成,可轻松实现应用程序的“一次编写,随处编译”。目前,Qt 主要由诺基亚的 Qt Development Frameworks (Qt 开发框架组) 负责开发和维护,用户涵盖全球 60 多个国家的 4400 多家厂商,如 Google、Adobe、IBM、华硕、CNTV、中国移动等,基于 Qt 的知名应用程序有 KDE、KOffice、Skype、Google Earth 等。

自 2008 年 6 月诺基亚并购奇趣科技后,Qt 在嵌入式移动平台上的发展大大提速。Qt 4.6 增加了 Symbian、Maemo 平台的支持,旋即发布的 QtMobility 开发包可提供各移动支持平台下的联系人、导航、网络连接等 API;Qt 4.7 引入了快速开发脚本语言 QML,为普通开发人员快速开发 Qt 应用程序提供了可能。而今集成了 Qt Creator 开发环境的 Nokia Qt SDK,加强了基于 Qt 开发 Symbian、Maemo/MeeGo/WinCE 应用程序的易用性,进一步为非专业人士投身移动平台应用程序开发降低了门槛。

在翻译本书的过程中,深感国内 Qt 技术力量的薄弱。在互联网如此发达的今天,除 Qt 中文论坛 (www.qtcn.org)、CuteQt 博客 (www.cuteqt.com) 和 CSDN 的 Qt 技术社区 (qt.csdn.net) 等少数网站外,大多处于停滞状态。Qt 中文论坛建立于 2005 年,面向广大初、中级 Qt 开发人员,是目前最为活跃的 Qt 综合技术中文讨论区;CuteQt 博客紧跟 Qt 的前沿技术领域和最新的 Qt 开发平台,由许多一线的 Qt 资深工程师负责维护和运行;CSDN 的 Qt 技术社区提供了许多权威资料。正是这些技术社区在不遗余力地积极推动着 Qt 技术在国内的发展。

由本书译者参与的《C++ GUI Qt 4 编程》(第二版)是第一本对 Qt 4 技术进行全面、系统介绍的中文权威译著。Jasmin Blanchette 和 Mark Summerfield 是该书的作者,也是 Qt 在线帮助文档的创建者和 Qt 开发人员。在书中,他们用许多示例程序和技术案例全面地介绍了 Qt 框架,使该书成为官方推荐的一本学习书籍,并作为诺基亚员工的 Qt 入门培训教材。正是如此,该书中文版自出版以来的两年内,已累计重印 6 次。

本书是《C++ GUI Qt 4 编程》(第二版)出版两年后 Mark Summerfield 的又一本学习 Qt 的里程碑级图书。在这本书中,不仅涵盖了《C++ GUI Qt 4 编程》(第二版)中部分过于高深和未能包含其中的内容,还有许多针对 Qt 技术底层细节的探讨,大多还没有在任何书籍中涉及过。因此,本书是近两年和未来一段时间内 Qt 高级技术的概括和预览,可帮助 Qt 编程人员切实提高他们使用 Qt 成就事业的能力。

在本书的翻译和审校过程中,我们坚持使用了“两译三审”的严苛做法,力求保证译稿质量,减少误译和纰漏。对于书中涉及的 Qt 和计算机技术词汇用语,则尽量与《C++ GUI Qt 4 编程》(第二版)一书的译法保持一致,避免读者产生困惑。同时,对于某些技术细节还向不少 Qt 一线开发工程师和原书作者 Mark Summerfield 做了求证。同时,结合英文原书的勘误信息,译者已将本书的相关代码更新至 2011 年 3 月。

本书的翻译和审校工作具体分工是:Qt 中文论坛管理员白建平 (XChinux) 负责本书的第 3 章、第 4 章、第 5 章和第 6 章翻译工作;西南科技大学的王军锋负责第 1 章、第 2 章、第 9 章和第 10 章的翻译;CuteQt 博客管理员 Shiroki 负责第 11 章和第 12 章的翻译;西北农林科技大学的闫锋欣负

责第7章、第8章、第13章以及书中剩余的前言、简介、精选参考书目和结束语等部分的翻译和质量控制工作。解放军装甲兵工程学院的吴迪(wd007)、高波和山东理工大学的戚彬负责了全书的终审和统稿。此外,我们还邀请了西安欧亚学院的周莉娜、赵延兵和韩二伟作为本书的外部审稿人。参与本书文字校对工作的还有朱加平、齐亮、王宁、赵拓和范文等人。

感谢电子工业出版社的编辑。他们对于计算机当前技术趋势的把握和战略眼光,为Qt系列图书的选题引进提供了许多方便,为译者提供了充足的工作时间,使译稿质量得到了最大限度的保证。正是他们对于图书选题的理解,才能为大家的Qt学习之路提供如此多的选择。

书中所用到的示例程序的源代码可从原书站点 [www. qtrac. eu/](http://www.qtrac.eu/) (英文) 下载,也可直接从 [www. qtcn. org/advqt/](http://www.qtcn.org/advqt/) (中文) 下载。

由于本版书中概念和术语数目繁多,并且许多概念和术语目前尚无公认的中文译法,加之译者水平所限,时间仓促,译稿中难免存在曲解或误解作者原意的地方,恳请读者谅解。读者也可以登录 [www. qtcn. org/advqt](http://www.qtcn.org/advqt) 参与讨论,我们也会在此及时更新本书的勘误信息。

译 者

2010 年 10 月



序 言

回想起 1991 年,我与 Haavard Nord 一起坐在挪威特隆赫姆公园中的长椅上。在我和 Haavard Nord 完成非服役服务的那段时间内,我俩一起为当地的一家医院开发一个超声波图像的存储和分析软件。这家医院使用了各种类型的计算机,因此,医院希望这个软件系统可以在 UNIX、Mac 和 Windows 平台上工作。这是一个巨大的挑战,我们调查市场,希望可以找到一些可以使用的类库。但对于找到的那些类库的质量,令我们担心不已。就在公园的那个长椅上,我们决定迎接这个挑战,提出我们自己的解决方案。

那个时候,我们都年轻、充满斗志且有些天真。讨厌浪费时间查找如何使用那些非直觉型的工具和库,我们决定改进这些工具和库。希望能稍微改善一下世界曾有的软件开发工作。我们的目标就是要让软件开发人员的生活变得更轻松些。众所周知,要实现这一目标,就要专注于软件开发中有趣的那一面,要富有创造性、编写出良好的代码。这样,我们就创造了第一个简单的 Qt 版本,并在数年后组建了 Trolltech(奇趣)公司。

至少,我们完成了一部分既定目标。自从 1995 年第一次发行 Qt 以来,它已取得了巨大的成功。

2008 年,Nokia 收购了 Trolltech,2009 年 4 月,对我而言,是时候要继续前进了。在公司里度过了 15 年又 27 天后,我不再是其中的一员了。

产品在优秀的人手中,团队的激情和繁重的工作一如既往。在 Nokia 的奇趣科技(Trolltech)依旧在努力确保 Qt 成为一流的、众望所归的框架。Lars Knoll(kHTML、鼎鼎大名的 WebKit 的鼻祖)今天正率领着近 150 名 Qt 专业工程师。作为授权协议的选择,Nokia 还新增了 LGPL 授权,这使 Qt 可被更多的开发人员所使用。

2009 年秋,我以荣誉嘉宾的身份受 Nokia 之邀参加了德国慕尼黑的 Qt 开发人员日(Qt Developer Days)活动。这个用户参与的会议(也在美国举办)是 Qt 爱好者的盛会,其规模正在逐年扩大。能够倾听来自全欧洲 Qt 用户间的探讨声,是一种很棒的感觉。我与许多开发人员进行了交流,他们都告诉我,在他们的软件开发工作中,Qt 的确与众不同。这让身为程序员的我感觉良好。

Qt 作为一个很棒的工具和类库,这仅仅是它取得成功的部分原因。用户还需要良好的文档、一些教程和书籍。毕竟,让开发人员生活更轻松才是目标。

这是我所深信不疑的,回溯到 2003 年也是如此。当时,我是 Trolltech 的总裁,负责文档的 Mark Summerfield 走进了我的办公室。他希望和 Jasmin Blanchette 一起写一本关于 Qt 的书。一部非常好的书,要由一个对产品知识无限熟悉且能够清晰、直观说明事物的人来撰写。当时,还有谁能够比 Qt 的文档负责人、同时也是最好的 Qt 开发人员的人,更适合这项工作呢?

最终的结果就诞生了一部关于 Qt 的伟大图书,并随后进行了更新和扩展。

Mark 现在又完成了另一项重要的工程。

在 Qt 编程人员的武器库中,就差一部有关 Qt 高级编程的书籍。我非常高兴,Mark 已经写完了这样的一部书。他是一名非常好的撰写技术书籍的作家,拥有撰写 Qt 编程书籍所应具有的全

部背景知识,是最具权威的人选。他专注于细节和清晰直观表达自己的能力总是令我印象深刻。也就是说,您一定会满意的!

在您手中(或通过屏幕阅读),您正紧握着一次扩充自己知识的极好机会,它可以让你用 Qt 做出许多超酷的东西。

编程快乐!

Eirik Chambe-Eng

于法国,南阿尔卑斯山

2009 年 12 月 24 日



前言

一段时间以来,我一直想写一本 Qt 书籍,一本能够涵盖《C++ GUI Qt 4 编程》一书中过于高深内容的书籍,尽管对一些读者来说,该书本身已经够有挑战性了。还有一些我打算涉及的专题材料(并非是比较难的)而是它们并没能包含进第一本关于 Qt 编程的书中。此外,从 Qt 庞大的规模上来看,也没有哪一本书能够对 Qt 所有的内容进行毫无偏颇的描述。毫无疑问,这为新技术文稿的撰写留下了空间。

这本书所做的就是从许多模块和各个方面的类中选择了一些内容,并展示该如何使用它们。这些所选择的主题都是我自己感兴趣的,同时好像也正是它们在 Qt 爱好者邮件群 qt-interest 中引起了许多讨论。这些主题中的一些主题还没有在任何其他书籍中涉及过,而另外一些主题则较为熟悉,比如,模型/视图编程。无论如何,我将尽量提供比其他可借鉴材料更为全面的内容。

因此,这本书的目的就是帮助 Qt 编程人员加深和拓宽他们的知识,提高他们使用 Qt 成就事业的能力。“高级”方面通常更多地是指能做到什么,而不是实现方法的手段。这是因为,正如常说的那样,Qt 让我们尽可能远离不相关的细节和潜在的复杂事物,提供易于使用的应用程序接口(API),从而只需简单、直接地使用就可以获得极好的效果。例如,我们将会看到:在不知道任何播放器工作原理的情况下,创建一个音乐播放器的过程;而所需要了解的仅仅是 Qt 所提供的那些高级 API。另一方面,即使对于高级 QtConcurrent 模块的用法,它所涵盖的对多线程的必要知识也都很具有挑战性。

这本书假设读者都具有基本的 C++ 编程能力,并且至少知道如何来创建基本的 Qt 应用程序——例如,已经读过一部好的 Qt 4 书籍,并有一定的工程实践经验。本书还认为,读者应该熟悉 Qt 的参考文档,至少能够使用它查询到感兴趣的类的 API。此外,一些章节会假设读者已经知道相关主题的基本知识——例如,第 1 章会假设读者已经知道一些 JavaScript 和 Web 编程的知识,在多线程的那些章节里,作者会假设读者能够理解线程的基本知识和 Qt 的线程类。所有这些假设都意味着,这本书将能够免于介绍那些 Qt 程序开发人员已经熟知的许多细节和类,比如布局的使用、动作的创建、信号和槽的连接等,从而可以让本书完全专注于那些读者不是很熟悉的知识。

当然,没有哪部单卷本书籍可以真正毫无偏颇地描述那 700 多个 Qt 公共类——在 Qt 4.6 中,几乎有 800 个,以及 100 多万字的 Qt 文档,所以本书也不会试图去那样做。相反,这本书为如何使用 Qt 最具强大功能的那些特征提供了一些说明和示例,用来补充参考文档而不是对它的重复。

本书在章节设计上,已尽可能做到内容完整,因而也就没有必要按照章节顺序自始至终地进行阅读。为了实现这一点,对于不同章节中要用到的那些特定技术,仅会在一个地方进行说明,而在其他地方则会使用交叉引用的方式给出。即使如此,如果你打算随机阅读一些零星章节,建议至少先对整本书做一个粗略的浏览,因为一些章节会专注于某个特定主题,而它又是其他主题必不可少的材料。同样,我将尽可能多地介绍那些完全来自 Qt 的 API 的小细节,以使本书的内容更为丰富,并在上下文中尽可能多地介绍那些特性,因而通篇会出现一些有用的信息。

与我之前那些书一样,本书中引用的代码段都是些“活代码”,也就是说,这些代码都是直接从例子的源文件中自动抽取并直接嵌入到送给出版商的 PDF 文件中的——因而就不会有剪切、粘贴方面的错误,而且可保证代码能够正常工作。这些例子可以从 www.qttrac.eu/aqbook.html 获得,基于 GPL(GNU General Public License,GNU 通用公共授权第 3 版)进行授权。本书将给出多达 25

个例子,分布在 150 多个 .hpp 和 .cpp 文件中,累计超过 20 000 行代码。尽管全部最为重要的代码段都在书中进行了引用和解释,但还有大量的细节无法在这本书内进行阐释,因此,建议下载这些示例并至少阅读一下那些你所特别感兴趣的例子的源代码。除了这些例子,本书还提供了一些包含有常用功能的模块。所有这些都用 AQP 命名空间来确保其重用性,开头的一些章节会将它们引进来,然后会在整本书中一直使用。

所有例子(除了最后一章中用到了 Qt 4.6 特性的那些例子)都用 Qt 4.5 和 Qt 4.6 在 Linux、Mac OS X 和 Windows 平台上进行了测试。使用 Qt 4.5 建立的那些应用程序将可以在 Qt 4.6 下不做修改而直接运行,对后续的其他 Qt 4.x 版本也可以运行,因为 Qt 在各个次要发行版中维持向后兼容。然而,对于这两个 Qt 版本之间的那些不同之处,本书会说明和解释与 Qt 4.6 相关的方法,而源代码部分会使用 `#if QT_VERSION`,以便可以用特定的版本或者最好的习惯来编译代码。一些例子或许可用于先前的 Qt 4.x 版本,特别是 Qt 4.4,且一些例子或许可以向后移植(backport)到更早的 Qt 版本——然而,这本书仅仅完全关注于 Qt 4.5 和 Qt 4.6,所以不会明确涉及到向后移植的问题。

本书给出了最好的 Qt 4.6 实践,尽管 Qt 4.6 比 Qt 4.5 包含更多的新特征,但对代码来说却没有太多不同。一个细微差别之处在于:Qt 4.6 有“退出”(quit)动作的快捷方式而 Qt 4.5 没有;源代码中,对于 Qt 4.6 会使用其快捷方式,而对于 Qt 4.5,则会用 `#if QT_VERSION` 表示与之功能相当的代码。更为重要的不同之处在于,Qt 4.6 引入了 `QGraphicsObject` 类,而且还在它与几何形状变化通信时改变了那些图形项(graphics item)的行为。我们会在某些地方说明这些不同之处,并在书中的代码段中给出 Qt 4.6 的方法,但是在源代码中,用 `#if QT_VERSION` 来说明如何使用 Qt 4.6 和 Qt 4.5 及其早期版本来完成同样的事情,并为两者选择最好的方法。在本书的最后一章,作为之前给出例子的转换,用三个例子中的两个来说明与 Qt 4.6 相关的那些特性,以及对 Qt 4.6 动画和状态机框架的应用。通过修改之前的例子,就更容易看出如何从传统的 Qt 方法过渡到新的框架下。

Qt 的下一个版本,Qt 4.7 将重点关注于稳定性、速度以及除 Qt Quick 之外的新技术(可提供一种使用类 JavaScript 语言创建 GUI 声明的方法),我们希望引入比之前发行版更少的新特性。尽管现在仍然有巨大的精力投入到 Qt 中,其范围也在不断扩大,但本书应当作为学习和使用 Qt 4.x 系列方面重要技术的一个有用资源,特别是对 Qt 4.5、Qt 4.6 和若干年后就要来临的那些后续版本来说。

致谢

我第一个要感谢的是我的朋友 Trenton Schulz, Nokia 公司 Qt 开发框架组(Qt Development Frameworks,之前的 Trolltech 公司)中的一名前任软件工程师,他目前是挪威计算中心(Norwegian Computing Center)的一名研究学者。事实证明,Trenton 是一名可靠的、富有远见的和挑战性的审稿人,他阅读仔细、标准严格,他提出的一些建议对改进本书相当有帮助。

接下来要感谢的是另外一位朋友,Jasmin Blanchette,他以前也是 Qt 开发框架中的一名软件工程师,与我一起合著了 *C++ GUI Programming with Qt 4*^① 一书,目前正在慕尼黑工业大学攻读博士

① 本书的中文译名是《C++ GUI Qt 4 编程》,已由电子工业出版社于 2008 年 8 月出版发行,详情请参阅 <http://www.phei.com.cn/bookshop/bookinfo.asp?bookcode=TP070380%20&booktype=main> 或 www.china-pub.com/42122——译者注。

学位。我们两个在前一段时间对这本书就形成了一致意见,而仅仅是因为工作的压力让他成为了一名出色的而且是苛刻的审稿人,而不是合著者。

我还要感谢很多那些工作(或任职)于 Qt 开发框架的人,他们阅读了该书的一些部分并提供了有益的反馈信息,还要感谢那些回答了技术问题的人,还有同时做了以上两件事的人。这些人包括:Andreas Aardal Hanssen(对图形/视图那几章给出了特别优秀的反馈和建议,并为我列出了离屏渲染方面的补充材料)、Andy Shaw、Bjørn Erik Nilsen、David Boddie、Henrik Hartz、Kavindra Devi Palaraja、Rainer Schmid(目前在 Froglogic)、Simon Hausmann、Thierry Bastian 和 Volker Hilsheimer。

意大利软件公司(www.develer.com)是一家很好的一个软件公司,为我提供了免费主机,让我能够在漫长的写作过程中安心完成这本书。他们的一些开发人员给了我有用的反馈,特别是早期章节中的一些例子。我特别感谢 Gianni Valdambri、Giovanni Bajo、Lorenzo Mancini(为我创建了资料库)和 Tommaso Massimi。

特别感谢初稿读者 Alexey Smirnov,他指出了一些错误,并鼓励我在一些网络示例中加入对网络代理的支持。

我还要感谢 Froglogic 的创始人,Reginald Stadlbauer 和 Harri Porten——他们提供给我的兼职顾问的工作,以帮我找到了写作这本书的时间,也同时向我介绍了一些编程技术,它们对我来说都是一些全新的想法。他们还把我变成了他们的 GUI 应用程序测试工具——Squish 的超级爱好者。

我的朋友 Ben Thompson 也应得到许多感谢,他帮我回忆起一些已经忘却的、可靠的数学概念,并且尤其要感谢他的耐心,一遍遍地向我解释这些数学概念直到我能够理解为止。

若没有 Qt,这本书(以及其他一些书)就不会成为现实。因此,我非常感谢 Qt 的创始人 Eirik Chambe-Eng 和 Haavard Nord,尤其要感谢 Eirik,他允许我在 Trolltech 的时候,把撰写我的第一本书作为日常工作,并且他还花费时间和精力来为这本书写了序言。

要特别感谢我的编辑,Debra Williams Cauley,相当独到地建议我优先撰写这本书,并且,她的支持和帮助让工作取得了实质性的进展。还要感谢 Jennifer Lindner 在书籍的结构和其他反馈方面做出的有效努力。还要感谢 Audrey Doyle,她对本书的出版管理工作认真负责,还要感谢审校读者 Barbara Wood,做出了那么好的审校工作。

我还要感谢我的妻子 Andrea,她与我一起经历了撰写过程中的坎坎坷坷,感谢她不朽的爱和无尽的支持!



目 录

第 1 章	混合桌面/Internet 应用程序	1
1.1	Internet 相关窗口部件	2
1.2	WebKit 的使用	11
第 2 章	声音和视频	33
2.1	QSound 和 QMovie 的使用	33
2.2	Phonon 多媒体框架	38
第 3 章	模型/视图表格模型	56
3.1	Qt 的模型/视图架构	56
3.2	用于表格的 QStandardItemModel	58
3.3	创建自定义表格模型	74
第 4 章	模型/视图树模型	85
4.1	用于树 QStandardItemModel 的用法	86
4.2	创建自定义树模型	100
第 5 章	模型/视图委托	122
5.1	与数据类型相关的编辑器	122
5.2	与数据类型相关的委托	124
5.3	与模型相关的委托	132
第 6 章	模型/视图中的视图	136
6.1	QAbstractItemView 子类	136
6.2	与模型相关的可视化视图	147
第 7 章	用 QtConcurrent 实现线程处理	162
7.1	在线程中执行函数	164
7.2	线程中的过滤和映射	172
第 8 章	用 QThread 实现线程处理	190
8.1	独立项的处理	190
8.2	共享项的处理	200
第 9 章	创建富文本编辑器	211
9.1	QTextDocument 简介	211
9.2	创建自定义的文本编辑器	212
9.3	一个单行的富文本编辑器	227
9.4	编辑多行的富文本	234
第 10 章	创建富文本文档	238
10.1	高质量地输出 QTextDocument 文件	239

10.2	创建 QTextDocument	241
10.3	输出和打印文档	246
10.4	绘制页面	251
第 11 章	创建图形/视图窗口	258
11.1	图形/视图架构	258
11.2	图形/视图窗口部件和布局	260
11.3	图形项简介	264
第 12 章	创建图形/视图场景	270
12.1	场景、项和动作	271
12.2	增强 QGraphicsView 的功能	289
12.3	创建可停靠的工具箱窗口部件	290
12.4	创建自定义图形项	294
第 13 章	动画和状态机框架	309
13.1	动画框架简介	309
13.2	状态机框架简介	312
13.3	动画和状态机的结合	316
结束语	324
精选书目	326



第1章 混合桌面/Internet 应用程序

- Internet 相关窗口部件
- Webkit 的使用

目前,无处不在的“云计算”,依靠网络驱动的手机,体积小巧的上网笔记本电脑和智能笔记本产品(更不用说 Google Doc 的文件存储系统了),再加上基于网络的零部署成本的应用程序,这一切都使桌面应用程序成为即将灭绝的恐龙——但它们对此却熟视无睹。

在我们抛弃 C++ 和 Qt 而转向网络程序,体验 JavaScript 和 HTML 所带来的微妙乐趣之前,回顾一下桌面应用程序所能带来的优势还是很有必要的。

- 可用性——在特定的负有关键任务的区域之外,我们相信很少(通常是因为不方便)会出现 Internet 不可用的情况,诸如因网络故障、ISP 错误等。此时,那些基于网络的应用程序将毫无用处^①。
- 资源获取途径——桌面应用程序可以不受任何限制地获取用户计算机上的所有资源,但基于网络的应用程序却会因为安全限制而不能发挥全部功能。
- 观感(look and feel)——桌面应用程序除了自身的菜单栏和工具条之外,没有多余的(让人迷惑不解的)浏览器菜单栏和工具条。它拥有自己的快捷键,并且不会与浏览器的快捷键相冲突。它的观感永远是在程序产生时设定好的,而基于 Internet 的应用程序的观感却会因为浏览器的改变而有所不同。
- 自定义窗口部件——桌面应用程序可以提供给用户一些具有常用功能的控件,并且,它的性能是网络应用程序无法比拟的。

在最为理想的情况下,我们能够拥有桌面应用程序所带来的全部优点,并且在网络可用时能够享受 Internet 所带来的好处。感谢 Qt 4.4 引入的 QtWebKit 模块,它能够实现我们的这一梦想。QtWebKit 可以创建在线(online)和离线(offline)都能使用的混合桌面/互联网应用程序。

与基于 Internet 的应用程序相比,Qt 的劣势主要在于其部署方面——它只能在用户的计算机上运行。如果部署消耗或带宽利用率必须为最小值时,可以采取一些办法来应对这种情况,例如,将大部分程序功能放在一些可以独立更新的小插件里。又或者,使用为 JavaScript (ECMAScript 语言)开发的 QtScript 模块或如果想使用不同的脚本语言的话,可以使用第三方的脚本模块——利用程序脚本来提供应用程序的大部分功能,再根据需求更新或添加独立脚本即可。还可以将尽可能多的功能放在服务器和网页脚本中,这样就可以节省很多为升级客户端而消耗的时间。

本章的重点在于 Qt 所提供的支持混合应用程序的关键技术。在 1.1 节中,我们将使用 Qt 4.4 中引入的便捷类 QNetworkAccessManager 来创建与互联网相关的窗口部件;在 1.2 节中,将使用 QtWebKit 模块,首先开发一个通用网络浏览器窗口部件——这借助于 QtWebkit 模块所提供的功能可以非常容易地实现它。然后,将使用刚才开发的通用浏览器组件,例如创建一个特定网站应用程序——利用 QtWebKit 读取在后台下载网页的 DOM (Document Object Model, 文档对象模型),这

^① 例如,可以参阅 opencloudcomputing.info/trends/cloud-computing-downtime 或云计算事件数据库 (Cloud Computing Incidents Database, CCID)。

样就可以从中为后续的开发工作提取信息。我们将看到如何把 Qt 窗口部件,甚至是自定义的窗口部件嵌入到网页中去,来提供一些使用标准 HTML 窗口部件无法实现的功能。

1.1 Internet 相关窗口部件

对 Internet 相关窗口部件(Internet-aware widget)的定义是:一种自动从 Internet 检索数据信息的窗口部件,它的构建可以是一次性事件,也可以是一种有规律性时间间隔的事件。

最简单地创建一个 Internet 相关窗口部件的方法是使用 QNetworkAccessManager 对象,构造出它的窗口部件子类。QNetworkAccessManager 的这些对象可以完成 HTTP(包括 HTTPS)的 HEAD、POST、GET 和 PUT 请求,并处理相关的 cookie(使用 QNetworkCookieJar)和身份验证(使用 QAuthenticator)。

我们将在本节介绍一个样例,它使用 QNetworkAccessManager 定时从 Internet 读取数据,还会介绍另外一个使用 QNetworkAccessManager 响应图片下载请求的样例。这两个示例完全可以说明如何使用 QNetworkAccessManager。图 1.1 给出了 QNetworkAccessManager 和外部网站之间的关系示意图。值得注意的是,QNetworkAccessManager 是 Qt 中 QtNetwork 模块的一部分,任何使用到该模块的应用程序都要在其 .pro 文件中加一行 QT += network。

本节中的例子是一个任务栏托盘图标程序。此程序主要用来显示那些常用控制操作(如音量控制)的图标,或用来提供内存的使用状态和当前的时间等信息。此处,我们将开发一个 Weather Tray Icon 应用程序(weathertrayicon)。它能从美国国家气象服务机构(U. S. National Weather Service, www. weather. gov)检索当前某个机场的天气信息(图标和数据),然后将相应的图标显示出来。

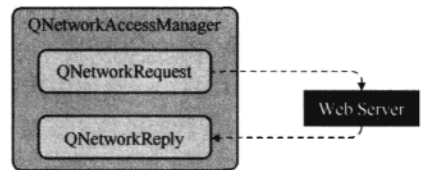


图 1.1 与外部站点通信的 QNetworkAccessManager

图 1.2 中左边的截图给出的是 Weather Tray Icon 应用程序和一个提示工具——其图标在提示工具的右下角;右边的截图显示的是该应用程序的上下文菜单,程序会每隔一个小时下载一次气象数据及相应的天气图标,还会根据下载的内容更新选定机场的天气情况。

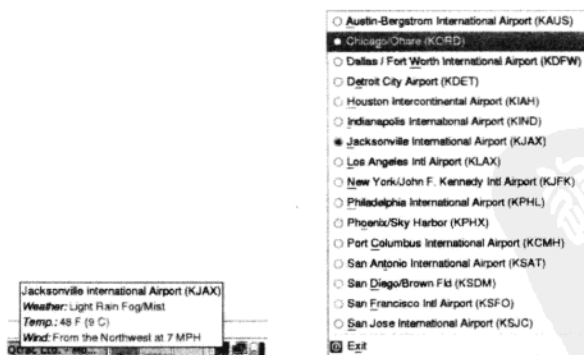


图 1.2 Weather Tray Icon 应用程序及其上下文菜单

像这样的任务栏托盘图标应用程序可以在所有支持 Qt 的桌面平台上运行。图 1.2 就是在基于 Linux 的 Fedora 系统运行 GNOME 桌面时的截图。在 Windows 和 Mac OS X 平台下,工具提示(tooltip)仅仅显示成纯文本,因为这两个平台不支持 Qt 工具提示的富文本格式(HTML)。当然,在 Mac OS X 下,图标仍然会像我们期待的那样显示在菜单栏上。

在本书中的大部分示例中,我们通常不会给出 `main()` 函数,因为它非常简单,并且经过了标准化。但在这个示例中,它与标准的 `main()` 函数稍微有点不同,所以这里将给出 `Weather Tray Icon` 应用程序的 `main()` 函数。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    app.setApplicationName(app.translate("main",
                                         "Weather Tray Icon"));
    app.setOrganizationName("Qtrac Ltd.");
    app.setOrganizationDomain("qtrac.eu");
    app.setQuitOnLastWindowClosed(false);
    if (int error = enableNetworkProxying())
        return error;

    WeatherTrayIcon weatherTrayIcon;
    weatherTrayIcon.show();
    return app.exec();
}
```

该函数以标准的 Qt 方式建立了一个 `QApplication` 对象。设定了应用程序的名称,这会在后面用到(如用于对话框的标题)可以通过 `QApplication::applicationName()` 来获取,也可以用组织的名称和定义域来设置它,这就意味着,我们可以在任何想创建 `QSettings` 对象的时候创建它,而无须给定任何参数。

这个函数有两个不同寻常的地方。第一,在最后的窗口已经关闭时,必须告诉 Qt 不要关闭应用程序。这是因为,正常情况下,一个托盘图标应用程序是没有窗口的(仅保留一个托盘图标),它所使用的任何窗口都是临时的(如一个工具提示或情景菜单),关闭它们不会导致程序的终止。

第二个方面是,自定义的 `enableNetworkProxying()` 函数的调用。我们在“网络代理的支持”的阴影部分中讨论过这个函数。如果它返回一个非零的错误代码,那么就意味着有错误出现,需要终止程序。

网络代理统一支持

对于能够直接连接到(如使用宽带调制解调器或路由器)Internet 的设备来说,本章所列举的这些例子都能够正常运行。然而,对于处于带有防火墙网络的那些内部设备来说,尤其是企业网络,这些示例可能无法连接到 Internet。大部分带有防火墙的网络都有某种能够提供网络连接的代理服务器。Qt 为这些代理提供了支持,所以我们已经为 `browserwindow`、`nyrbviewer`、`rsspanel` 和 `weathertrayicon` 等示例提供了代理支持,这可以通过在 `main()` 函数内部调用自定义的 `enableNetworkProxying()` 函数来实现。

`enableNetworkProxying()` 函数利用 `AQP::OptionParser`(由本书在 `option_parser` 中并且使用 `AQP` 命名空间的示例提供)解析用来建立代理的命令行参数。启用代理的应用程序支持的命令行选项有:

<code>-h</code>	<code>--help</code>	显示帮助信息并终止
<code>-H</code>	<code>--host = STRING</code>	主机名,如 <code>www.example.com</code>
<code>-P</code>	<code>--password = STRING</code>	密码
<code>-p</code>	<code>--port = INTERGER</code>	端口号,如 <code>1080</code>
<code>-t</code>	<code>--type = STRING</code>	代理类型(<code>http</code> , <code>socks5</code> ;默认为 <code>socks5</code>)
<code>-u</code>	<code>--username = STRING</code>	用户名

只有在指定了主机名后才能在 `enableNetworkProxying()` 函数中建立代理。以下是其代码,解析器的类型是 `AQP::OptionParser`。

```
if (parser.hasValue("host")) {
    QNetworkProxy proxy;
    proxy.setType(parser.string("type") == "socks5"
        ? QNetworkProxy::Socks5Proxy
        : QNetworkProxy::HttpProxy);
    proxy.setHostName(parser.string("host"));
    if (parser.hasValue("port"))
        proxy.setPort(parser.integer("port"));
    if (parser.hasValue("username"))
        proxy.setUser(parser.string("username"));
    if (parser.hasValue("password"))
        proxy.setPassword(parser.string("password"));
    QNetworkProxy::setApplicationProxy(proxy);
}
```

如果已经指定了主机,那么就可以利用已给出的主机、默认或给定的代理类型和用户拥有的其他信息来建立代理。在这个示例中,为整个程序设定了全局代理。也可以使用 `QAbstractSocket::setProxy()` 分别为每一个 socket 建立代理。

程序会提供多种格式的天气数据,但这里选择使用 XML 格式。此格式非常简单,实际上只包含了成对出现的键值列表,键为一个标签名,其值是位于开始和关闭标签之间的文本,例如:

```
<weather>Fair</weather>
<temperature_string>49 F (9 C)</temperature_string>
<temp_f>49</temp_f>
<temp_c>9</temp_c>
<wind_string>From the Northeast at 5 MPH</wind_string>
<visibility_mi>9.00</visibility_mi>
<icon_url_base>http://weather.gov/weather/images/fcicons/</icon_url_base>
<icon_url_name>nskc.jpg</icon_url_name>
```

当程序第一次启动时,它将机场设定为用户上一次设定,或者是当程序第一次运行时默认指定的某个值。然后,利用 `QNetworkAccessManager` 取得天气数据。气象数据包含的两个元素为: URL 和一个表示机场当前天气状况图标文件名。应用程序使用第二个 `QNetworkAccessManager` 来获取图标,并把它作为任务栏托盘里的程序图标。实际上,程序会将图标隐藏起来以节省带宽,稍后我们将会看到这种情况。

```
class WeatherTrayIcon : public QSystemTrayIcon
{
    Q_OBJECT
public:
    explicit WeatherTrayIcon();

private slots:
    void requestXml();
    void readXml(QNetworkReply *reply);
    void readIcon(QNetworkReply *reply);
    void setAirport(QAction *action);

private:
    ...
    QMenu menu;
```

```

QNetworkAccessManager *networkXmlAccess;
QNetworkAccessManager *networkIconAccess;
QString airport;
QCache<QUrl, QIcon> iconCache;
int retryDelaySec;
};

```

稍后会给出并解释所有的方法,包括那些没有给出的私有方法,但现在我们要先对该类的一些私有成员进行说明。`airport` 字符串的值为当前选定的机场,如“Chicago/Ohare (KORD)”。`iconCache` 拥有 `QUrl` 键和指向 `QIcon` 的指针。其余的成员变量我们会在讨论那些用到它们的函数时再给出说明。

`QCache` 类使用“成本”(cost)模式来缓存各个项。可以缓存的项的最大值默认为 100(成本项的总数一般等于或小于最大设定值)。默认情况下,每个项的成本值都是 1,因此,当在没有修改该最大值或自行设定我们自己的项的成本值时,能够缓存高达 100 个项。在增加一个新项时,如果该项的成本超过了最大的成本值,那么就会移除一个或多个最近访问的项,直到项的成本总数小于或等于最大值。

`QCache` 在后台利用 `QHash` 通过键值进行快速轮询。然而,出乎意料的是,`QHash` 不能将 `QUrl` 存储为键值,因为 Qt 没有提供 `qHash(QUrl)` 函数^①。这可以通过添加一行代码来做到。

```
inline uint qHash(const QUrl &url) { return qHash(url.toString()); }
```

在这里,我们仅把任务传递给内置的 `qHash(QString)` 函数。

我们现在已经为查看这些方法做好了准备,就先从构造函数开始。

```

WeatherTrayIcon::WeatherTrayIcon()
: QSystemTrayIcon(), retryDelaySec(1)
{
    setIcon(QIcon(":/rss.png"));
    createContextMenu();

    networkXmlAccess = new QNetworkAccessManager(this);
    networkIconAccess = new QNetworkAccessManager(this);
    connect(networkXmlAccess, SIGNAL(finished(QNetworkReply*)),
            this, SLOT(readXml(QNetworkReply*)));
    connect(networkIconAccess, SIGNAL(finished(QNetworkReply*)),
            this, SLOT(readIcon(QNetworkReply*)));
    QTimer::singleShot(0, this, SLOT(requestXml()));
}

```

我们给定程序一个初始图标,在等待第一个天气图标下载时,可以先暂时使用它。然后创建一个带有动作的上下文菜单,用它来改变机场和终止程序。

绝大多数的构造函数都专门用于通过创建两个 `QNetworkAccessManager` 来建立 Internet 访问路径。其中一个用来获取天气数据,另一个用来获取当前天气情况相关的图标。它们分别使用单独的网络路径管理器,以便使其独立工作,对于这两条路径,仅创建一个单独的信号-槽连接,因为我们感兴趣的是每个下载是什么时候完成的。

最后,我们使用一个单计时器来调用 `requestXml()` 槽。这种方法使用 `networkXmlAccess` 网络路径管理器来取得当前机场的天气数据。

我们已经简单、直接地调用了 `requestXml()`,但考虑到编程风格,还是建议尽量少用在各个构造函数中构造一个对象的“create”方法,也尽量少用单触发器来调用那些构造后初始化方法(post-construction initializing method)。这样做可确保在调用初始化方法时,对象已经完全构造好了。这意味着初始化方法可以访问任何的成员变量或方法,但在对象的构造过程中这样做则不一定安全。

^① Qt 4.7 已经计划提供一个 `qHash(QUrl)` 函数。

在介绍 `requestXml()` 槽之前,先简要看一下上下文菜单是如何建立的,以弄明白用户可以设置机场的哪些详细信息必须下载下来。

```
void WeatherTrayIcon::createContextMenu()
{
    QStringList airports;
    airports << "Austin-Bergstrom International Airport (KAUS)"
    ...
    << "San Jose International Airport (KSJC)";
    QSettings settings;
    airport = settings.value("airport", QVariant(airports.at(0)))
        .toString();

    QActionGroup *group = new QActionGroup(this);
    foreach (const QString &anAirport, airports) {
        QAction *action = menu.addAction(anAirport);
        group->addAction(action);
        action->setCheckable(true);
        action->setChecked(anAirport == airport);
        action->setData(anAirport);
    }
    connect(group, SIGNAL(triggered(QAction*)),
        this, SLOT(setAirport(QAction*)));
    menu.addSeparator();
    menu.addAction(QIcon(":/exit.png"), tr("E&xit"), qApp,
        SLOT(quit()));
    AQP::accelerateMenu(&menu);
    setContextMenu(&menu);
}
```

这里通过硬编码(hard-coded)列表给出了机场的名字,但它们本可以简单地从文件或资源文件中读取出来(如果想列出所有美国的机场,可以简单地将其分组,如可以通过将州作为在顶部菜单项,机场作为子菜单项的方法)。使用 `QSettings` 设定当前机场,在初次运行程序时,默认为列表中的第一个机场。

为每一个机场创建一个 `QAction`,并为当前机场选择一个相匹配的动作。把所有的机场动作添加到一个 `QActionGroup` 中。默认情况下,`QActionGroup` 拥有设置为 `true` 的唯一属性。这就保证了它的每个动作都拥有的是单选按钮而不是复选按钮,每次也只能选中一个机场。

还向程序添加了一个退出(exit)动作,并为其指定了一个专有的键盘加速键。然后,调用 `AQP::accelerateMenu()` 为尽可能多的机场提供键盘加速键,并将之前建立的菜单设定为应用程序的上下文菜单。如果是在 Mac OS X 平台上编写程序,正常情况下,调用 `AQP::accelerateMenu()` 是无效的,因为该平台不支持加速键。关于自动设定键盘加速键的更多内容,请参阅“键盘加速键”的阴影部分。

要把每个动作连接到 `setAirport()` 槽,并以某种方式参数化每个槽的调用,以便让槽知道选择的是哪个机场。一种简单的办法是在槽内部调用 `QObject::sender()`,以查看出是哪个动作调用了它,然后提取该动作的文本来判别出所选择的机场。另外一种方法是使用 `QSignalMapper`。但在这种情况下,还有一种更为容易的解决方案——连接 `QActionGroup`,而不是每一个机场的 `QAction`。`QActionGroup::triggered()` 信号会把相关的 `QAction` 作为它的参数。

```
void WeatherTrayIcon::requestXml()
{
    QString airportId = airport.right(6);
    if (airportId.startsWith("(") && airportId.endsWith("))") {
        QString url = QString("http://www.weather.gov/xml/"
            "current_obs/%1.xml").arg(airportId.mid(1, 4));
        networkXmlAccess->get(QNetworkRequest(QUrl(url)));
    }
}
```

给定机场的 XML 格式天气数据会存储到一个文件中,文件的名字与机场的 4 个字母代码相对应。我们已经将这个代码放在每个机场名字后面的括号里了,可以使用基本的 QString 方法来提取它。一旦有了需要的 URL,就可以使用 XML 网络路径管理器对其做出 GET 请求。当某个请求变成了指向 QNetworkReply 对象的指针,它就能接收到返回的值。这些对象发射出反映进度情况的信号,例如,downloadProgress() 和 uploadProgress(),当请求完成后,响应对象会发出一个 finished() 信号。

初始化该请求的网络管理器会发射一个 finished() 信号,且因为这个信号是我们连接到的唯一信号,就不必去过多关注监测进程,这也是忽略 QNetworkAccessManager::get() 方法返回值的原因。一旦下载结束(无论成功与否),前面给出的信号-槽连接都可以确保用一个指向响应对象的指针来调用 readXml() 方法,因为这个指针是该方法的唯一参数。

```
void WeatherTrayIcon::readXml(QNetworkReply *reply)
{
    if (reply->error() != QNetworkReply::NoError) {
        setToolTip(tr("Failed to retrieve weather data:\n%1")
            .arg(reply->errorString()));
        QTimer::singleShot(retryDelaySec * 1000,
            this, SLOT(requestXml()));
        retryDelaySec <= 1;
        if (retryDelaySec > 60 * 60)
            retryDelaySec = 1;
        return;
    }
    retryDelaySec = 1;
    QDomDocument document;
    if (document.setContent(reply))
        populateToolTip(&document);
    QTimer::singleShot(60 * 60 * 1000, this, SLOT(requestXml()));
}
```

如果请求失败,将错误信息显示在托盘图标的工具提示上,然后在延迟时间之后再次尝试。延迟时间的值保存在私有 retryDelaySec 变量中,初始值为 1(必须用 1000 乘以此值,因为 QTimer::singleShot() 的超时设定是以毫秒为单位的)。对于接连的失败,我们把延迟时间的间隔翻倍,将整数左移一位(bit)就可以让它的值翻倍。在第二次失败之后,经过两秒钟后再次尝试,然后是四秒钟,以此类推。照这种情形,经过数次失败后,时间间隔就会超过一个小时,这时要把它重新设置成一秒钟。

我们让重试间隔不断变化,这样可以避免误认为是服务攻击而遭到拒绝,避免变成“不走运的”请求/失败循环。

只要请求成功,就把 retryDelaySec 重置为 1,然后对 XML 数据进行解析。QNetworkReply 是 QIODevice 的一个子类,它不仅能发射网络进程信号,还能发射如 readyRead() 之类的信号。像其他 QIODevice 文件一样,我们可以从中读取数据,默认返回一个 QByteArray。QDomDocument::setContent() 方法可以从 QByteArray、QString 或 QIODevice 中读取 XML,所以可以直接把 QNetworkReply 传递给它进行解析。

键盘加速键(针对非 Mac OS X 平台)

键盘加速键对于那些打字非常快和那些不能或不愿意使用鼠标的用户来说非常重要。它们是形如 Alt + x 的键序列(此处, x 通常是一个字母或数字),用户可以按下这样的组合键从菜单栏中拉下一个菜单(例如, Alt + F 对应 File 菜单)。只要菜单出现,就可以只按下某某菜单项中带有下划线的字母或数字来选择该项,所以用户按下 Alt + F、N 就可以建立一个新文件。

在对话框中,加速键用来变换键盘焦点所在的特定窗口部件,如一个以 Total: 出现的标签拥有的键盘加速键就是 Alt + T,按下它就可以把键盘焦点转移到标签的伙伴(buddy)窗口部件上。同样,复选项和单选按钮通常也都有加速键,当按下加速键时,它们的状态会发生改变。

在屏幕显示和编码的过程中,使用大写字母作为加速键(如 Alt + E)是一种常见的方法(键盘加速键和键盘快捷方式是有区别的,后者是与特定动作关联的任意键序列,例如,Ctrl + N 用来建立一个新文件)。

对于短小的菜单和只包括少量窗口部件的对话框来说,手动设定加速键(通过在文本中包含一个“&”符号)是非常简单的。但如果菜单项或窗口部件的个数超过 15 个,就很难决定哪种加速方法更好些了。理想的解决办法是让计算机替我们来决定加速键,这也是本书中使用的方法。

在 aqp 目录中,alt_key. {hpp,cpp} 模块提供了一个 API,kuhn_munkres. {hpp,cpp} 模块提供了一种快速计算最佳结果的算法,它运行起来比所有的原生算法更简单。例如,要为一个主窗口的菜单提供加速键,可以把下面这行代码放到靠近主窗口构造函数的末尾处:

```
AQP::accelerateMenu(menuBar());
```

这就是所有需要做的,Kuhn-Munkres 算法用来计算最优的加速键,并且不影响所有已有加速键的项,如我们想让某个菜单选项或标签拥有其特定的加速键,但不必关心它具体是什么。如果是对话框,可以使用一个同样简单的方法,把下面这行代码放到对话框的构造函数的末尾处即可:

```
AQP::accelerateWidget(this);
```

加速键函数不能用于隐藏窗口部件,如那些在一个标签页里面,但并不显示出来的窗口部件。这种情况仍然可以处理,我们可以通过使用一个或多个 AQP::accelerateWidgets() 函数的调用,每次给它一个特定的有效的窗口部件列表。

如果 QDomDocument::setContent() 方法返回值为 true,则解析成功,用新数据填充工具提示——这个进程也可能涉及到下载一个新的图标。如果解析失败,保留原来的工具提示。最后,设定一个单定时器,以便可在一个小时后再次重复整个过程。

QDomDocument 类是 Qt 的 QtXml 模块的一部分,因此可以使用它,但必须在应用程序的 .pro 文件里面加上 QT += xml 这一行。

程序的控制流程,从在 XML 网络接入管理器上初始化 GET 请求,到下载 XML 和图标,然后每小时重复这个下载过程,图 1.3 给出了整个流程的示意图。

为保证完整性,我们将分成三个部分来说明私有的 populateToolTip() 方法,然后是它的两个私有帮助方法。

```
void WeatherTrayIcon::populateToolTip(QDomDocument *document)
{
    QString tooltipText = tr("<font color=darkblue>%1</font><br>")
        .arg(airport);
    QString weather = textForTag("weather", document);
    if (!weather.isEmpty())
        tooltipText += tooltipField("Weather", "green", weather);
    ...
}
```

此处提取了我们想要包含在工具提示中的文本元素,而忽略了每个元素的大部分代码,原因是它们的风格都与 weather 元素的代码相似(或者非常相似)。

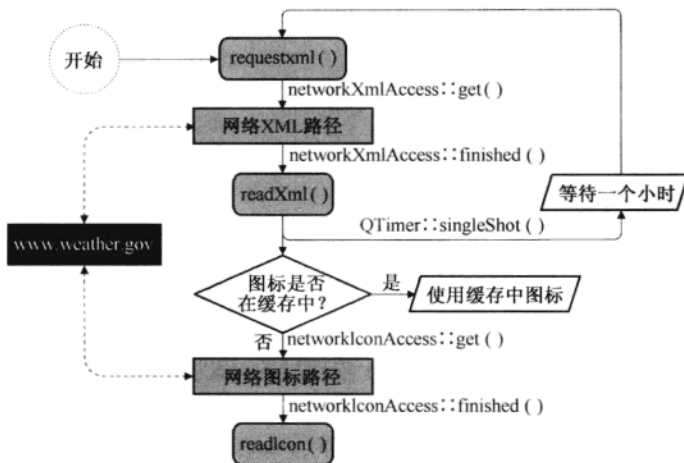


图 1.3 Weather Tray Icon 程序的控制流程

私有的 `textForTag()` 帮助方法用来从任意给定标签中取得文本。这样是可行的, 因为对于天气数据来说, 每个标签都是唯一的, 并且不包含任何嵌套标签。

```

QString iconUrl = textForTag("icon_url_base", document);
if (!iconUrl.isEmpty()) {
    QString name = textForTag("icon_url_name", document);
    if (!name.isEmpty()) {
        iconUrl += name;
        QUrl url(iconUrl);
        QIcon *icon = iconCache.object(url);
        if (icon && !icon->isNull())
            setIcon(*icon);
        else
            networkIconAccess->get(QNetworkRequest(url));
    }
}

```

与主要天气状况关联的图标由两个 XML 元素确定, 即 `icon_url_base` 和 `icon_url_name`。我们会试图提取这两个元素的文本并在它们之外建立一个 URL。然后尝试使用 URL 作为键, 从缓存中取得图标。如果缓存中不存在任何对应此键的项, 则 `QCache::object()` 方法返回 0。

如果从缓存中检索到了一个 `QIcon`, 那么就使用它——实际上我们获取的是一份副本(它的开销较小而且很有价值, 前者是因为 Qt 使用的写时复制(copy-on-write)机制, 后者是由于 `QCache` 可以随时删除项)。否则, 我们可以使用图标网络路径管理器来下载图标。如果一个图标开始下载, 需要早点建立信号-槽连接, 以保证在下载结束时调用 `readIcon()` 槽(前面简短地介绍过)。

```

#ifdef Q_WS_X11
    toolTipText = QTextDocumentFragment::fromHtml(toolTipText)
        .toPlainText();
#endif
setToolTip(toolTipText);
}

```

遗憾的是, 只有 X11 支持在托盘图标工具提示中使用 Qt 富文本(HTML), 因此, 对于 Windows 和 Mac OS X 系统, 需要把 HTML 格式的工具提示文本转换成纯文本。要做到这一点, 可以先用静态的 `QTextDocumentFragment::fromHtml()` 方法获得一个 `QTextDocumentFragment`, 然后再使用 `QTextDocumentFragment::toPlainText()` 产生纯文本。使用 `QTextDocumentFragment` 比手动转换要方便很

多,因为它不仅能将 HTML 实例转换成合适的 Unicode 字符,剔除 HTML 标签,还能智能到足以把 `
` 转换成换行符。

```
QString WeatherTrayIcon::textForTag(const QString &tag,
                                     QDomDocument *document)
{
    QDomNodeList nodes = document->elementsByTagName(tag);
    if (!nodes.isEmpty()) {
        const QDomNode &node = nodes.item(0);
        if (!node.isNull() && node.hasChildNodes())
            return node.firstChild().nodeValue();
    }
    return QString();
}
```

对于较小的 XML 文件使用 `QDomDocument` 是比较理想的,因为它会对整个文件进行解析,把所有数据保存在内存中,还提供了多种方便的访问方法。

在这里,首先要取得一个所有使用特定标签的 `QDomNode` 列表。如果列表非空,就取得第一个节点。在天气数据中,每一个标签都是唯一的,因此,每个给定标签只能对应唯一的节点。在 DOM API 中,标签间的文本保存在子节点中,所以它们可以通过取得节点的第一个子节点来得到,把子节点转换成一个文本节点,然后取得它的文本数据。例如, `node.firstChild().toText().data()`。令人欣慰的是,Qt 提供了一种快捷方式——`QDomNode::nodeValue()` 方法,它返回一个节点与类型相关的 (type-specific) 字符串,此字符串在文本节点中就是文本自身。

```
QString WeatherTrayIcon::toolTipField(const QString &name,
                                     const QString &htmlColor, const QString &value, bool appendBr)
{
    return QString("<i>%1:</i>&nbsp;<font color=\"%2\">%3</font>%4")
        .arg(name).arg(htmlColor).arg(value)
        .arg(appendBr ? "<br>" : "");
}
```

私有 `toolTipField()` 帮助方法可以让我们解析出每一行工具提示文本的格式。这稍微缩短并简化了 `populateToolTip()` 的代码,并使得后续对格式进行修改变得容易些。

如果一个图标必须下载, `QNetworkReply` 一旦就绪,前面建立的信号-槽连接就要保证对 `readIcon()` 槽的调用。我们将分两部分来介绍这个槽。

```
void WeatherTrayIcon::readIcon(QNetworkReply *reply)
{
    QUrl redirect = reply->attribute(
        QNetworkRequest::RedirectionTargetAttribute).toUrl();
    if (redirect.isValid())
        networkIconAccess->get(QNetworkRequest(redirect));
}
```

这个方法会在图标的下载请求完成时得到调用。首先,要检查已接收到一个某种类型的重定向,看它是否是我们期望的反馈结果。如果是的话,启动一个新的 GET 请求,以使用重定向的目标 URL 来取得图标数据。出于对安全的考虑, `QNetworkAccessManager` 不会自动重定向,但在这里,我们选择相信站点。如果安全问题是重点,就需要检查重定向的 URL。例如,它是来自相同的主机,而且不含一些怀有恶意 JavaScript。通常情况下,如果没有强制重定向,会得到一个无效的重定向 `QUrl`,这样就可以继续读取反馈的数据。

```
else {
    QByteArray ba(reply->readAll());
    QPixmap pixmap;
    if (pixmap.loadFromData(ba)) {
        QIcon *icon = new QIcon(pixmap);
        setIcon(*icon);
    }
}
```

```

        iconCache.insert(reply->request().url(), icon);
    }
}

```

如果反馈结果不是重定向,可得到一个图标数据,或者是一个错误。读取得到的所有数据并放入到 `QByteArray` 中,然后把数据提供给 `QPixmap`。如果 `QPixmap::loadFromData()` 方法返回 `false`,那么就表示图标数据是不完整的、损坏的,或者是一个不可识别的格式,也或者因为有网络错误发生而没有取得任何数据。无论上述哪种情况发生,放弃了从网络上获取图标的尝试,则当前图标就不会改变。

如果下载成功,把 `QPixmap` 转换成 `QIcon`,并把它设定为托盘图标。然后,通过添加由 URL 键入的图标到图标缓存中,出于对安全的考虑,不要在缓存中保存超过 100 个图标;不要下载一个缓存中已存在的图标,这没有什么必要。

```

void WeatherTrayIcon::setAirport(QAction *action)
{
    airport = action->data().toString();
    QSettings settings;
    settings.setValue("airport", airport);
    requestXml();
}

```

无论用户什么时候从上下文菜单中选择一个新的机场,都会调用这个槽。取得机场的名字,把这个机场设定为新的默认值,然后调用 `requestXml()` 使程序为新选择的机场获取最新的天气数据。

需要注意的是,程序在终止时,不会保存任何设定,而是在设定发生改变的时候,才保存它(在这个示例中只有一个设定)。这种方法的好处在于,无论是在程序运行的时候,还是在程序面临预期之外崩溃的时候,设定总是最新的。但是不足之处在于,保存设定的代码会蔓延到所有地方,这使得维护变得很容易出错。

我们现在已经完成了对一个小程序的介绍,它利用高级和易于使用的 `QNetworkAccessManager` 类实现基本的 Internet 下载功能。除了 `Weather Tray Icon` 应用程序之外,本书还包括了 `RssPanel` 程序(`rsspanel`)示例,如图 1.4 所示。

`RssPanel` 应用程序的特征是一个 Internet 相关的 `RssComboBox`,它可以从一个 RSS (Really Simple Syndication, 聚合内容) 自动填充内容。RSS 是一个 XML 文件,如果给予它一个合适的 URL,它会定期自动更新。我们将不再分析这个示例的代码,因为它的结构与 `Weather Tray Icon` 示例非常相似。然而,它或许为我们创建自己的 Internet 相关窗口部件提供了一个容易的开端。它还使用一个 `QXmlStreamReader` 子类,而不是 `weathertrayicon` 示例^①使用的 `QDomDocument` 来对下载的 RSS 数据进行解析。

本章的剩余部分将继续介绍从 Internet 获取数据这一主题,只是改为使用 `WebKit` 来显示数据(如 HTML 页面),并对下载的数据进行处理。

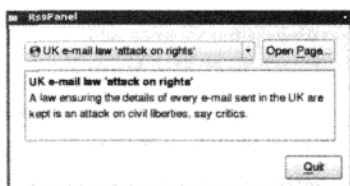


图 1.4 `RssPanel` 应用程序

1.2 WebKit 的使用

`WebKit` 是一个开源网络内容渲染和编辑引擎,它最初由 KDE (“K” Desktop Environment, “K” 桌面环境)的开发者们创建。现在,`WebKit` 是很多网络浏览器的基础,包括 Google 的 `Chrome`、KDE 的

① 最终,为 Qt 提供 DOM 和 SAX 解析器的 `QtXml` 或许会被废弃掉,已经内置于 `QtCore` 的更快速的 `QXmlStreamReader` 和 `QXmlStreamWriter` 类则更受欢迎。

Konqueror 和 Mac OS X 的 Safari,许多支持网络的移动设备也使用它。WebKit 致力于与标准兼容,它支持所有的标准网络技术,包括 HTML5、SVG(Scalable Vector Graphics,可伸缩矢量图形)、CSS(Cascading Style Sheets,层叠样式表,包括 CSS 3 网络字体)和 JavaScript。Qt 的 QtWebKit 模块为 WebKit 提供了一个 Qt 风格的交互界面,这让使用 Qt 的编程人员能使用 WebKit 的功能,它还提供了许多额外的自身功能。要使用 QtWebKit 模块,必须要在程序的 .pro 文件中加入一行:QT += webkit。

表 1.1 中列举了最重要的 QtWebKit 类,图 1.5 显示了它们之间的一些关系。

表 1.1 Qt 中主要的 WebKit 类

类	说 明
QWebElement	一个类,用来获取和编辑一个 QWebFrame 的 DOM 元素,这是通过它的一个类似 JQuery 的 API 实现的(Qt 4.6 以后支持)
QWebFrame	一个数据对象,用来代表一个网页中的一个框架
QWebHistory	一个类,用来代表和给定的 QWebPage 相关的历史访问链接
QWebHistoryItem	一个对象,用来代表在 QWebHistory 中一个访问过的链接
QWebPage	一个数据对象,用来代表一个网页
QWebSettings	一个数据对象,用来保存给定的 QWebFrame 或 QWebPage 的设定
QWebView	一个窗口部件,用来可视化一个 QWebPage

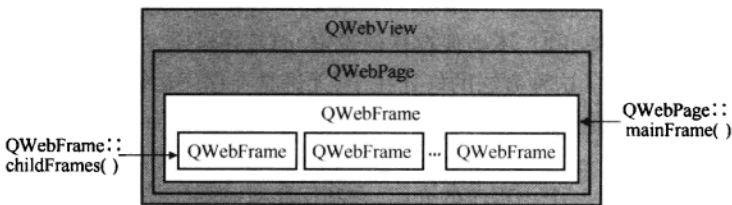


图 1.5 上下文中的 QtWebKit 类

注意,QtWebKit 中唯一的窗口部件是 QWebView。例如,它所包括的 QWebPage 和(一个或多个)QWebFrame 都是数据类。使用 QWebPage 可以在后台下载并处理网络内容,然后可以任意选择一种方式把结果反馈到用户界面,我们将在 1.2.2 节中看到这些相关内容。

现在,我们已经对 QtWebKit 模块有了一个初步的印象,下面看一些使用它的示例。在 1.2.1 节中,我们将创建一个网络浏览器窗口组件;1.2.2 节将创建一个具体站点的应用程序,它使用这个浏览器窗口组件并在后台读取和处理网页;1.2.3 节说明如何把 Qt 窗口部件和自定义窗口部件嵌入到一个网页中。

1.2.1 一个通用网络浏览器窗口组件

在接下来的两个小节中,我们将在 WebKit 的帮助下开发两个混合桌面/Internet 应用程序。在本小节中,我们将创建一个如图 1.6 所示的浏览器窗口组件(browserwindow),后面的样例会用到它。

浏览器窗口支持标准的浏览器特征:前进、后退、重新加载、取消加载、缩放、打开给定的页面,并且具有返回到浏览器历史记录中某个特定的页面的能力。它还有一个上下文菜单和一个工具栏(可以隐藏起来)。

另外,在定义一个自定义的 DEBUG 符号时(例如,通过在 .pro 文件中添加 `DEFINES += DEBUG`),浏览器窗口的上下文菜单会额外显示一个选项 Inspect,它启动后会调用图 1.7 中显示的 WebKit Web Inspector。这是一个有用的调试工具,它能提供关于一个网页的丰富信息,包括页面

的 DOM(Document Object Model, 文档对象模型)、使用的资源(例如, 样式表、图像和 JavaScript 脚本), 以及它们的大小、载入时间等更多的信息。

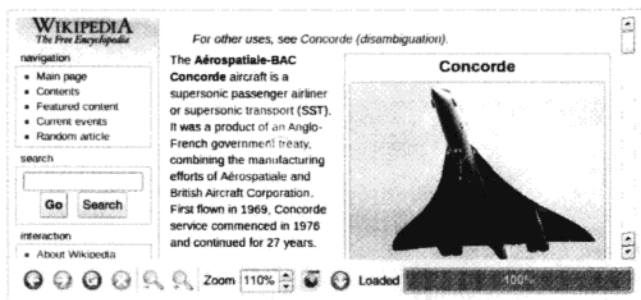


图 1.6 浏览器窗口组件

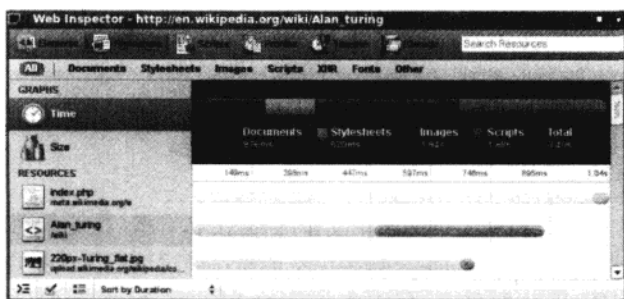


图 1.7 Web Inspector

从 Qt 4.6 开始, 我们可以更方便地调用 Web Inspector——创建一个 QWebInspector 对象, 给它一个 QWebPage, 然后调用它的 show() 方法就可以了。

为了让 Web Inspector 可用(并为了让 QWebInspector 类在 Qt 4.6 中可工作), 我们必须开启 QWebSettings::DeveloperExtrasEnabled 网络设置。完成之后, 还要在应用程序的 main() 函数中做一些其他各种类型的设置, 然后把改变后的设置应用于全局的 QWebSettings 对象, 下面这些从 main() 函数中提取的代码片段就显示了这一过程。

```
QWebSettings *webSettings = QWebSettings::globalSettings();
webSettings->setAttribute(QWebSettings::AutoLoadImages, true);
webSettings->setAttribute(QWebSettings::JavascriptEnabled, true);
webSettings->setAttribute(QWebSettings::PluginsEnabled, true);
webSettings->setAttribute(QWebSettings::ZoomTextOnly, true);
#ifdef DEBUG
webSettings->setAttribute(QWebSettings::DeveloperExtrasEnabled,
                        true);
#endif
```

全局的 QWebSettings 对象的设置由应用程序中所有的 QWebPage 和 QWebView 对象继承, 当然, 如果愿意的话, 也可以逐个地重写每一个 QWebPage 和 QWebView 的设置。Qt 4.5 引入了 QWebSettings::ZoomTextOnly 属性, 它影响了伸缩因子。通过把这个属性设定为 true, 可以确保图像不会被缩放(因此, 如果是像素图的话, 它们将不会产生变形)。当用户进行缩放操作时, 只有文本会被缩小或放大。

直到本书成文时, Qt 文档还没有具体指定网络设置的默认值, 因此, 这个值可能会因为平台与 Qt 4.x 版本的不同而有所差异。我们总是可以利用 QWebSettings::testAttribute() 检查特定的设定值, 它获取一个属性的枚举值, 返回一个 bool 型变量。

为了对浏览器窗口的 API 有一个概览,我们将介绍头文件中类定义的公有部分和受保护部分。

```
class BrowserWindow : public QFrame
{
    Q_OBJECT

public:
    explicit BrowserWindow(const QString &url=QString(),
                           QWebPage *webPage=0, QWidget *parent=0,
                           Qt::WindowFlags flags=0);

    QString toHtml() const
    { return webView->page()->mainFrame()->toHtml(); }
    QString toPlainText() const
    { return webView->page()->mainFrame()->toPlainText(); }

signals:
    void loadFinished(bool ok);
    void urlChanged(const QUrl &url);

public slots:
    void load(const QString &url);
    void setHtml(const QString &html) { webView->setHtml(html); }
    void showToolBar(bool on) { toolBar->setVisible(on); }
    void enableActions(bool enable);

protected:
    void focusInEvent(QFocusEvent*) { webView->setFocus(); }
```

大部分功能,特别是关于工具栏和上下文菜单动作的,是由私有槽(在此未列出)提供的,我们会在接下来的代码片段中遇到它们时再讨论。如果浏览器窗口通过编程方式指定的方法(通过对其调用 `QWidget::setFocus()`)来获取焦点,焦点将传递给 Web 视图,要确保这一点,需要重新实现 `QWidget::focusInEvent()` 方法。`BrowserWindow` 这个类也包含一些私有变量(未列出),提供了对它里面的窗口部件的访问。

由于 `QWebView` 提供了超出我们想象的功能,`BrowserWindow` 相当小巧,它的大部分代码在构造函数和调用的创建方法处。下面是它的构造函数。

```
BrowserWindow::BrowserWindow(const QString &url, QWebPage *webPage,
                              QWidget *parent, Qt::WindowFlags flags)
    : QFrame(parent, flags)
{
    setFrameStyle(QFrame::Box|QFrame::Raised);

    webView = new QWebView;
    if (webPage)
        webView->setPage(webPage);
    load(url);

    createActions();
    createToolBar();
    createLayout();
    createConnections();
}
```

我们把浏览器窗口定义成一个 `QFrame` 的子类,这样就可以给它赋一个框架。这对用户是非常有用的,因为网页通常包括自身的窗口部件(按钮、行编辑器,等等),通过为浏览器窗口赋予框架可以明确网页和它所嵌入的应用程序之间的边界。

如果类的客户端想要通过其自己的 `QWebPage`,我们允许它这样做,否则,`QWebView` 会为它自己创建一个 `QWebPage`。如果我们想使用 `QWebPage` 子类的话,这种做法将很有效,我们将在题为“把 Qt 窗口部件嵌入到网页中”的阴影部分中看到 `QWebPage` 子类。

`createActions()` 方法有点不同,因为我们只要自己建立一些动作就可以了。以下代码段是从它里面提取的,它显示了 `zoomOutAction` 动作的建立,忽略了 `zoomInAction`、`setUrlAction` 和 `historyAction` 的建立过程,因为它们的建立方式都是相同的。

```
void BrowserWindow::createActions()
{
    zoomOutAction = new QAction(QIcon(":/zoomout.png"),
                                tr("Zoom Out"), this);
    zoomOutAction->setShortcuts(QKeySequence::ZoomOut);
    ...
    QList<QAction*> actions;
    actions << webView->pageAction(QWebPage::Back)
    << webView->pageAction(QWebPage::Forward)
    << webView->pageAction(QWebPage::Reload)
    << webView->pageAction(QWebPage::Stop)
    << zoomOutAction << zoomInAction << setUrlAction
    << historyAction;
#ifdef DEBUG
    actions << webView->pageAction(QWebPage::InspectElement);
#endif
    AQP::accelerateActions(actions);
    webView->addActions(actions);
    webView->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

我们建立了一系列想让浏览器窗口拥有的动作,如果已设定 `DEBUG`,包括 `inspect` 动作,然后还包括在可能的地方使用 `QWebView` 预定义的动作。在非 `Mac OS X` 平台上,可使用 `AQP::accelerateActions()` 提供键盘加速键(也就是带有下画线的字母,而不是类似于 `Ctrl + X` 的键盘快捷方式)。然后,将这些动作附加到 `QWebView`,让它使用这些动作来提供一个上下文菜单。

值得注意的是,我们不需要建立信号-槽连接或者为 `QWebView` 提供的动作准备对应的槽,因为它们都是内建的。我们并没有说明 `createToolBar()` 方法,因为它对于 `C++/Qt` 应用程序来说几乎就是标准的方法。然而,如图 1.8 所示(对比图 1.6),工具栏在 `Mac OS X` 上的显示效果与在其他平台上显示的是不同的。这是因为,通常在 `Mac OS X` 上的工具按钮会显示成一个图标加上其下面的文字。为保证整个工具栏的显示效果都相同,我们会为缩放微调框加上标签,为进度条的下面加上标签,而不是像在其他平台上那样加在左面。为得到这些差异,在 `browserwindow/browserwindow.cpp` 文件的代码中使用了一些 `#ifdef`,以便仅在 `Mac OS X`^① 上才会定义 `Q_WS_MAC`。

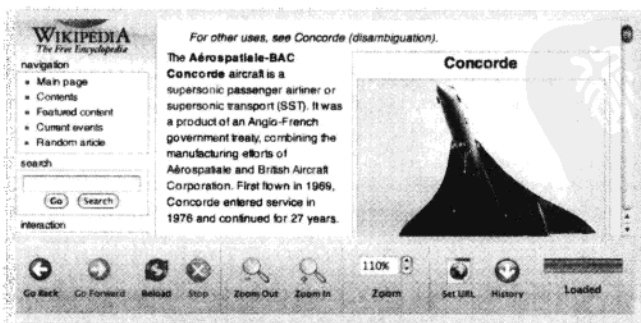


图 1.8 Mac OS X 平台上的浏览器窗口组件

① 像在本书的“前言”部分提到的那样,可以从 www.qtrac.eu/aqpbook.html 获取全部源代码。

`createLayout()` 方法比较短小并且是标准函数, 所以会跳过它来继续看 `createConnections()` 的代码。

```
void BrowserWindow::createConnections()
{
    connect(webView, SIGNAL(loadProgress(int)),
            progressBar, SLOT(setValue(int)));
    connect(webView, SIGNAL(urlChanged(const QUrl&)),
            this, SLOT(urlChange(const QUrl&)));
    connect(webView, SIGNAL(loadFinished(bool)),
            this, SLOT(loadFinish(bool)));
    connect(setUrlAction, SIGNAL(triggered()), this, SLOT(setUrl()));
    connect(historyAction, SIGNAL(triggered()),
            this, SLOT(popUpHistoryMenu()));
    connect(zoomOutAction, SIGNAL(triggered()),
            this, SLOT(zoomOut()));
    connect(zoomInAction, SIGNAL(triggered()), this, SLOT(zoomIn()));
    connect(zoomSpinBox, SIGNAL(valueChanged(int)),
            this, SLOT(setZoomFactor(int)));
}
```

建立前三个信号-槽连接的目的是, 可以保持对加载过程和页面改变的追踪, 这样就可以让进度条和进度标签保持更新。其余的连接是为了处理我们自己建立的动作, 并相应用于与缩放微调框的交互而设的。

现在, 我们已经看到了足够多的内容, 从而对浏览器窗口有了一个大致了解, 下面我们来看看

```
void BrowserWindow::load(const QString &url)
{
    if (url.isEmpty())
        return;
    QString theUrl = url;
    if (!theUrl.contains("://"))
        theUrl.prepend("http://");
    webView->load(theUrl);
}
```

如果给定的 URL 字符串为空, 不做任何处理, 否则, 如果没有指定, 预先挂出 `http://` 以示礼貌, 则告诉 Web 视图执行加载(使用 `QString` 而不是 `QUrl`, 因为前者较后者更容易为字符串正确添加)。

```
void BrowserWindow::setUrl()
{
    load(QInputDialog::getText(this, tr("Set URL"), tr("&URL:")));
}
```

如果用户调用了设置 URL 的动作, 程序就会调用这个方法。如果用户取消了设置, 程序就会把一个空的 URL 传递给 `load()` 方法, 它将不会做任何事情, 也不会带来什么不利之处。

```
void BrowserWindow::urlChange(const QUrl &url)
{
    emit urlChanged(url);
    progressLabel->setText(tr("Loading"));
}
```

无论网页的 URL 什么时候发生了变化(不管是用户点击了链接、使用了设置 URL 的动作, 或者是使用了历史记录), 都会调用这个槽。我们发射出自己的 `urlChanged()` 信号, 为 `BrowserWindow` 的用户提供方便, 然后更新进度标签来提示加载操作已经开始执行。

```
void BrowserWindow::loadFinish(bool ok)
{
    emit loadFinished(ok);
    progressLabel->setText(ok ? tr("Loaded") : tr("Canceled"));
}
```

当加载操作结束时,不管它是否成功,程序都会用 bool 型的 ok 变量的值来调用 loadFinish() 这个槽。接下来再次发射一个信号,为 BrowserWindow 的用户提供方便,然后再次更新进度标签以反映当前情况。

不需要担心如何让进度条保持更新,因为我们在构造函数的末尾处把 Web 视图的 loadProgress() 信号连接到了进度条的 setValue() 槽。

```
void BrowserWindow::setZoomFactor(int zoom)
{
    webView->setZoomFactor(zoom / 100.0);
}
```

如果用户操作了缩放微调框,程序就会调用 setZoomFactor() 这个槽,Web 视图中的文字会相应地得到缩放(如果想要让图像缩放,必须将 QWebSettings::ZoomTextOnly 属性设为 false)。

```
const int ZoomStepSize = 5;
void BrowserWindow::zoomOut()
{
    zoomSpinBox->setValue(zoomSpinBox->value() - ZoomStepSize);
}
```

zoomOutAction 动作是连接到 zoomOut() 这个槽的。程序中也有相似的执行放大的动作和对应的槽,虽然我们没有对它们进行介绍。当这些槽被调用时,调用 setValue() 槽可以让选值微调框发射一个 valueChanged() 信号,由于我们前面看到的信号-槽连接的缘故,这会导致调用 setZoomFactor() 槽。

```
void BrowserWindow::enableActions(bool enable)
{
    foreach (QAction *action, webView->actions())
        action->setEnabled(enable);
    toolBar->setEnabled(enable);
    webView->setContextMenuPolicy(enable ? Qt::ActionsContextMenu
                                   : Qt::NoContextMenu);
}
```

在某些情况下,嵌入了浏览器窗口的应用程序不想让用户使用除了简单的查看和与当前页面进行交互之外的功能。enableActions 方法使得禁止或启用浏览器窗口的动作成为可能。

```
const int MaxHistoryMenuItems = 20;
const int MaxMenuWidth = 300;
void BrowserWindow::popUpHistoryMenu()
{
    QFontMetrics fontMetrics(font());
    QMenu menu;
    QSet<QUrl> uniqueUrls;
    QListIterator<QWebHistoryItem> i(webView->history()->items());
    i.toBack();
    while (i.hasPrevious() &&
           uniqueUrls.count() < MaxHistoryMenuItems) {
        const QWebHistoryItem &item = i.previous();
        if (uniqueUrls.contains(item.url()))
            continue;
        uniqueUrls << item.url();
        QString title = fontMetrics.elidedText(item.title(),
                                                Qt::ElideRight, MaxMenuWidth);
        QAction *action = new QAction(item.icon(), title, &menu);
        action->setData(item.url());
        menu.addAction(action);
    }
    AQP::accelerateMenu(&menu);
}
```



```
if (QAction *action = menu.exec(QCursor::pos()))
    webView->load(action->data().toUrl());
}
```

当调用这个方法时,程序会弹出一个菜单,菜单的项与用户已访问的网页相对应。链接数据是从 QWebView 的 QWebHistory 获得的。这样就取得了一个 QWebHistoryItem 列表,它的每一项都拥有网页的标题、网页的 URL、页面的图标(如果网页的服务器上没有可用的图标,Qt 会提供一个默认图标,以及一些其他信息)。

菜单以反向顺序显示了链接,也就是说,最近访问的在最顶端显示,最早访问的在最底端显示。它还对所显示的项数附加了限制,并剔除掉重复内容——这意味着访问链接不是严格按顺序排列的。一些页面的标题很长,在这种情况下,我们使用 QFontMetrics::elidedText() 从右端开始对它进行省略(也就是说,去掉超出的文本,并用一个省略号来代替)。通过把 Qt::ElideLeft 或者 Qt::ElideMiddle 作为 QFontMetrics::elidedText() 方法的第二个参数,也可以实现从左边开始省略,或者从中间开始省略。

前面我们曾经提到过,在 Qt 4.7 出现之前,Qt 没有提供 qHash(QUrl) 功能,所以我们不能够像所期望的那样把 QUrl 存储到 QSet 中。因为 QSet 是按照 QHash 的方式实现的,这个问题的解决方法是增加一行 qHash(QUrl) 代码,它和我们之前在 Weather Tray Icon 应用程序中所使用的完全相同。

如果用户取消了菜单(例如,通过按下 Esc 或点击其他地方),QMenu::exec() 将返回 0;否则它将返回与用户所选择的菜单项对应的 QAction。如果返回了一个 QAction,就从它的数据中提取出 URL。一旦有了 URL,我们就可以请求 Web 视图加载相应的页面。

现在已经完成了对窗口浏览器组件的介绍。我们还可以为它增加一些其他的标准浏览器特性,但由于 WebKit 已经提供了必需的功能,所以其中的一些是容易实现的。例如,可以基于 QWebView::findText() 增加一个搜索文本功能,基于 QWebFrame::print() 或者 QWebFrame::render() 增加一个打印页面功能。

在接下来的两小节中,我们将把这个浏览器窗口作为两个混合桌面/Internet 应用程序的基础。还将学习如何隐蔽的下载后台的网络内容,如何将 JavaScript 引入到网页以从中提取信息,还有如何增强浏览器窗口,让它可以无缝地显示用户可以与之交互的标准和自定义的 Qt 窗口部件。

1.2.2 创建特定网站的应用程序

如果人们经常访问某一特定网站,那么根据他们的需求,创建一个面向特定网站的应用程序能为他们提供更多的方便和额外的功能。这类应用程序的危险在于,它们很容易在网站中发生变化,但与强大的易用性所能节省的时间相比,这些变化就微不足道了。特别是当网站拥有大量用户时,这种程序更能发挥其效用。当然,我们可以控制这些改变,使其只影响后台使用的 JavaScript,而完全不会影响网站的源代码。

创建某个网站专用的应用程序还能确保用户只通过这个客户端软件就可以访问该网站。苹果公司的 iTunes Music Store 就是个很好的例子,它(在本书成文时)无法在标准的网络浏览器上使用。

在本小节中,我们将介绍 Books Viewer(nyrbviewer)应用程序的 New York Review,如图 1.9 所示。这个应用程序使用上一小节中的浏览器窗口组件显示来自 NYRB(New York Review of Books, 纽约书评)的网页。它还提供了用来显示主题以及所选主题文章的组合框列表,这使得该应用程序具有出众的易用性,使用它比在网络浏览器中浏览站点要方便得多。它可以让用户轻松地查看现有的主题和文章,并能快速地选择要阅读的文章。



图 1.9 Books Viewer 的 New York Review

它的大部分功能已经包含在浏览器窗口组件中了,因此我们只需要关注如何用正确的数据填充组合框,并让它们运行。先来看一下构造函数:

```
const QString NYRBUrl("http://www.nybooks.com");
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    createWidgets();
    createLayout();
    issueLinkFetcher = new LinkFetcher(NYRBUrl,
        scriptPathAndName("fetch_issue_links.js"), this);
    articleLinkFetcher = new LinkFetcher(NYRBUrl,
        scriptPathAndName("fetch_article_links.js"), this);
    createConnections();
    AQP::accelerateWidget(this);
    issueComboBox->setFocus();
    issueLinkFetcher->load(NYRBUrl);
    setWindowTitle(QApplication::applicationName());
    QTimer::singleShot(1000 * 60, this, SLOT(networkTimeout()));
}
```

我们以常用的方法建立窗口部件和布局。AQP::accelerateWidget() 函数使用 QObject::findChildren() 来查找所有该主窗口子部件的 QLabel 和 QAbstractButton (以及它们的子类,除了 QToolButton),并为它们设定键盘加速键。

此构造函数的一个特别之处在于,它建立了两个 LinkFetcher 对象。这两个对象会在后台建立 QWebPage 对象和再调用数据对象 QWebPage——让它们的 QWebPage 下载一个特定的页面。当在页面中建立这两个对象时,引入它们的 JavaScript,这样做有助于提取相对链接。它们的网站 URL 用来把所有相对链接转换为绝对链接,如把 articles/22273 转换成类似于 http://www.nybooks.com/articles/22273 的绝对链接。

一旦创建并连接了链接读取器,就可以让 issueLinkFetcher 下载 NYRB 网站的主页,利用 fetch_issue_links.js 脚本提取所有主题的连接;这些连接将用来填充主题复选框。无论什么时候选择主题,articleLinkFetcher 都会下载当前主题所在的页面,并利用 fetch_article_links.js 脚本提取所有主题文章的链接地址;这些地址将用来填充文章复选框。我们将在此小节的后半部分对 LinkFetcher 类进行说明。

因为我们想让脚本更具灵活性,所以创建了 `scriptPathAndName()` 方法,将马上对此进行说明。

在代码的最后部分,我们建立了一个单触发计时器(single shot timer)。它将在一分钟超时并调用 `networkTimeout()` 槽,这样的话,如果 Internet 不可用,它将返回一个说明网络不可用的信息。

```
QString MainWindow::scriptPathAndName(const QString &filename)
{
    QString name = filename;
    QString path = AQP::applicationPathOf() + "/";
    if (QFile::exists(path + name))
        return path + name;
    return QString(":/%1").arg(name);
}
```

这个方法会在包含应用程序可执行文件的目录中查找脚本,如果找到了脚本,就返回链接检查器所要读取的完整路径和名称。如果文件系统中没有保存脚本,就直接返回,把嵌入到可执行文件目录中的脚本作为资源。这意味着,应用程序会默认使用自己的嵌入式脚本,但如果要改变这个脚本(如网站正在改版),只需要简单地将更新好的脚本放入到可执行文件的目录中,程序就会自动查找到更新后的文件,并会优先使用此文件,而不是应用程序的嵌入式脚本(当然,这涉及到安全问题,我们更建议采用一个二进制文件格式,这个格式要带有检查机或其他方法以保证本地安装的脚本合法,这些内容都超出了本书的讨论范围)。

`aqp. {hpp,cpp}` 模块的 `AQP::applicationPathOf()` 函数返回应用程序的路径(以 `QString` 的形式),如果把子目录当做参数,那么返回的是可执行文件目录的子目录路径。我们不能直接使用 `QApplication::applicationDirPath()`,因为它没有考虑一个特殊情况,即可执行文件或许在另外一个目录中,这取决于它是处于发行状态还是处于正在开发状态(例如,在 Windows 平台的开发期间,可执行状态文件通常在 `debug` 或是 `release` 子目录中)。

```
const QString InitialMessage(
    QObject::tr("Attempting to connect to the network..."));
const QString FailMessage(
    QObject::tr("No issues or articles available"));

void MainWindow::networkTimeout()
{
    const QString text = browser->toPlainText().trimmed();
    if (text == InitialMessage || text == FailMessage)
        browser->setHtml("<h3><font color=red>Failed to connect "
            "to the network</font></h3>Perhaps the proxy "
            "settings are wrong, or maybe a proxy is needed. "
            "Try:<br><tt>nrybviewer --help</tt>");
}
```

应用程序启动一分钟后会调用这个槽。如果没有建立 Internet 连接,浏览器的文本将与 `InitialMessage` 或 `FailMessage` 相同,因此,我们试图为用户提供一些帮助(参见题为“支持网络代理”的阴影部分)。

在头文件中有一个私有变量, `namesForUrlsForIssueCache` :

```
QHash<int, QMap<QString, QString> > namesForUrlsForIssueCache;
```

这个缓存键是组合框中主题项的索引位置,它的值是映射,这些映射的键值为 URL,其值为文件名。我们将在介绍主窗口的方法时介绍它的使用方法,现在需要注意的是,链接获取器能为链接返回它们所在页面的 URL-名称映射(之所以没有使用 `QCache`,是因为我们不想去掉缓存数据)。

我们将不再介绍 `createWidgets()` 和 `createLayout()` 方法,因为它们是标准的 C++/Qt 方法,而仅给出 `createConnections()` 的介绍。


```

comboBox->clear();
comboBox->addItem(statusText);
QMapIterator<QString, QString> i(namesForUrls);
i.toBack();
while (i.hasPrevious()) {
    i.previous();
    comboBox->addItem(i.value(), i.key());
}
if (comboBox->count() > 1)
    comboBox->setCurrentIndex(1);
}

```

populateIssueComboBox() 和 populateArticleComboBox() 两个槽都调用了这个方法。

首先,清除组合框中原有的内容,把状态文本作为它的第一项。然后在传递进来的 URL-主题名映射(如果用来填充文章组合框,则是 URL-文章名映射)上反向迭代,用链接获取器返回这些映射。对于 NYRB 网站的主题和文章,其 URL 用日期进行编码,例如,/contents/20090115,我们按照从离现在最近到最远的顺序进行迭代。对于主题,它的名称是简单的美式格式的日期(例如,“Jan 15, 2009”),把这些内容添加为组合框的项文本,把 URL 添加为项数据。对于文章,它的名称是实际的文章标题。

如果存在多个项,也就是说,至少有一条主题或一篇文章,把第一条主题(最新的)或文章设定为当前项,依次调用 currentIndexChanged() 槽或 currentArticleIndexChanged() 槽。

```

void MainWindow::currentIssueIndexChanged(int index)
{
    articleComboBox->clear();
    if (index == 0) {
        articleComboBox->addItem(tr("- no issue selected -"));
        return;
    }
    if (namesForUrlsForIssueCache.contains(index))
        populateArticleComboBox();
    else {
        articleComboBox->addItem(
            tr("+ fetching the list of articles +"));
        browser->setHtml(tr("<h3><font color=red>"
            "Fetching the list of articles...</font></h3>"));
        QString url = issueComboBox->itemData(index).toString();
        articleLinkFetcher->load(url);
    }
}

```

如果用户选择了一条新主题,应用程序会从清除文章组合框开始执行。如果选择了一条主题,检查是否已经下载了它的 URL-文章名映射;如果已经完成了下载,调用 populateArticleComboBox() 就可以了,它一般从缓存中检索数据。

如果没有可用于新选择主题的缓存,则可以向文章组合框和浏览器窗口中增加表示状态的文本。然后提取主题的 URL(保存在主题组合框项的数据中),告诉文章链接获取器下载主题页面,并从中提取文章链接。由于存在前面创建的信号-槽连接,一旦下载结束,程序就会调用 populateCache() 槽。

```

void MainWindow::populateCache(bool ok)
{
    if (!ok || issueComboBox->count() == 1) {
        articleComboBox->setItemText(0,
            tr("- no articles available -"));
        browser->setHtml(tr("<h3><font color=red>%1</font></h3>"
            ".arg(FailMessage));
    }
}

```

```

        return;
    }
    QTextDocument document;
    QMap<QString, QString> namesForUrls =
        articleLinkFetcher->namesForUrls();
    QMapIterator<QString, QString> i(namesForUrls);
    while (i.hasNext()) {
        i.next();
        document.setHtml(i.value());
        i.setValue(document.toPlainText());
    }
    namesForUrlsForIssueCache[issueComboBox->currentIndex()] =
        namesForUrls;
    populateArticleComboBox();
}

```

如果下载 URL-文章名称失败,在文章组合框和浏览器窗口显示一些文本信息,告诉用户下载失败。如果下载成功,从文章链接获取器处检索 URL-文章名称映射。然后把修改过的映射添加到缓存,调用 `populateArticleComboBox()`。

文章名要以纯文本形式输入到 `QComboBox` (不能显示富文本)。不是每次都去建立并销毁 `QTextDocumentFragment` (即 `i.setValue(QTextDocumentFragment::fromHtml(i.value()).toPlainText())`),而是建立一个单独的 `QTextDocument`,并在每次迭代时设置它的 HTML 文本,然后再调用它的 `toPlainText()` 方法。

```

void MainWindow::populateArticleComboBox()
{
    int index = issueComboBox->currentIndex();
    if (index > 0)
        populateAComboBox(tr("- no article selected -"),
            namesForUrlsForIssueCache[index], articleComboBox);
}

```

只有在缓存已经得到了请求数据时才会调用这个方法,因此,如果选择了一条主题,那么它的链接就存在于缓存中了。用前面提到的 `populateAComboBox()` 方法填充组合框,传递给它一个状态文本,并将此文本添加为第一项,从缓存中提取当前主题的 URL-文章名映射,把它填充到需要填充的组合框。如果用户已经在主题组合框中选择了第一项 (“no issue selected” 项),那么不做任何处理。

```

void MainWindow::currentArticleIndexChanged(int index)
{
    if (index == 0)
        return;
    QString url = articleComboBox->itemData(index).toString();
    browser->load(url);
    browser->setFocus();
}

```

当用户在文章复选框中选择一个新项 (不同于第一个 “no article selected” 项) 时,从组合框项的数据中提取与当前文章标题对应的 URL,告诉浏览器加载相应的页面。赋予浏览器窗口焦点 (它实际会转到浏览器窗口的 `QWebView` 中),这样用户就可以使用键盘的 `up` 和 `down` 键实现滚动了。

到目前为止,我们已经看到了 Books Viewer 应用程序的 New York Review 的完整实现过程 (除了头文件, `createWidgets()` 和 `createLayout()` 方法)。让我们回顾一下 `LinkFetcher` 类,看看它是如何在后台下载页面并提取链接的。以下是它的头文件:

```

class LinkFetcher : public QObject
{
    Q_OBJECT

```

```

public:
    explicit LinkFetcher(const QString &site_,
        const QString &scriptOrScriptName_, QObject *parent=0);

    void load(const QString &url);
    QMap<QString, QString> namesForUrls() const
    { return m_namesForUrls; }
    void clear() { m_namesForUrls.clear(); }

signals:
    void finished(bool);

public slots:
    void addUrlAndName(const QString &url, const QString &name);

private slots:
    void injectJavaScriptIntoWindowObject();
    void fetchLinks(bool ok);

private:
    QWebPage page;
    QMap<QString, QString> m_namesForUrls;
    const QString site;
    const QString scriptOrScriptName;
};

```

这个类用 QWebPage 数据类加载指定的 URL。然后用指定的 JavaScript 脚本从页面提取相关链接，然后用每个链接的 URL 和名称填充 m_namesForUrls。

我们可以让所有的 QObject 变得能让 JavaScript 访问。这意味着，引入到下载页面的 JavaScript 不仅可以访问页面元素（使用 HTML 文件对象模型），还可以访问任何可用的 QObject。尤其是 JavaScript 可以在一个 QObject 的公共槽中调用方法，读取它所有的 QObject 的属性。在这种情况下，程序需要传递一个引用（reference）给链接获取器实例，这样，引入的 JavaScript 脚本就可以和我们编写的 C++ 代码通信，这些是通过调用连接获取器的公共槽完成的。

```

LinkFetcher::LinkFetcher(const QString &site_,
    const QString &scriptOrScriptName_, QObject *parent)
    : QObject(parent), site(site_),
      scriptOrScriptName(scriptOrScriptName_)
{
    QWebSettings *webSettings = page.settings();
    webSettings->setAttribute(QWebSettings::AutoLoadImages, false);
    webSettings->setAttribute(QWebSettings::PluginsEnabled, false);
    webSettings->setAttribute(QWebSettings::JavaEnabled, false);
    webSettings->setAttribute(QWebSettings::JavascriptEnabled, true);
    webSettings->setAttribute(QWebSettings::PrivateBrowsingEnabled,
        true);

    connect(page.mainFrame(), SIGNAL(javascriptWindowObjectCleared()),
        this, SLOT(injectJavaScriptIntoWindowObject()));
    connect(&page, SIGNAL(loadFinished(bool)),
        this, SLOT(fetchLinks(bool)));
}

```

既然在后台下载页面，并且只关注页面的链接，那么就需要改变 QWebPage 的页面设定。关闭图像下载、插件、Java、打开 JavaScript 和 Settings::PrivateBrowsingEnabled，以防止 QWebPage 记录任何访问历史或是存取任何网页图标，因为这两者对我们都毫无用处，它或许还会消耗一些内存和 CPU 时间。

无论何时开始下载一个页面，清除页面 QWebFrame 的 JavaScript 窗口对象，为所有新页面框架包含的 JavaScript 做好准备。因为我们需要将自己的 JavaScript 引入到每个已下载页面的主框架的窗口对象，所以必须确保我们的 JavaScript 会在清除所有 JavaScript 窗口对象时重新引入。此处给出的第一个信号-槽连接完成了此项任务。

第二个信号-槽连接确保了只要一下载页面,代码就尝试提取页面的链接。

```
void LinkFetcher::injectJavaScriptIntoWindowObject()
{
    page.mainFrame()->addToJavaScriptWindowObject("linkFetcher",
                                                    this);
}
```

`QWebFrame::addToJavaScriptWindowObject()` 方法可以添加任何 `QObject` 到一个 `QWebFrame` 的 JavaScript 窗口对象。第一个参数是 JavaScript 可用对象的名称(在此例中,是 `linkFetcher`), `QObject` 的第二个参数是实际对象的一个引用。此时,它是 `LinkFetcher` 类的一个实例。

```
void LinkFetcher::load(const QString &url)
{
    clear();
    page.mainFrame()->load(QUrl(url));
}
```

当链接获取器加载一个 URL 时,它会从清除 `m_namesForUrls` 映射开始,并让 `QWebPage` 的主框架加载页面。一旦加载结束(因为前面已建立的信号-槽连接),就调用 `fetchLinks()` 槽。

```
void LinkFetcher::fetchLinks(bool ok)
{
    if (!ok) {
        emit finished(false);
        return;
    }
    QString javaScript = scriptOrScriptName;
    if (scriptOrScriptName.endsWith(".js")) {
        QFile file(scriptOrScriptName);
        if (!file.open(QIODevice::ReadOnly)) {
            emit finished(false);
            return;
        }
        javaScript = QString::fromUtf8(file.readAll());
    }
    QWebFrame *frame = page.mainFrame();
    frame->evaluateJavaScript(javaScript);
    emit finished(true);
}
```

如果加载失败,那么通知所有连接对象。否则,检查 `QString scriptOrScriptName` 私有成员变量是真的拥有一个脚本,还是仅有脚本的名称;如果是后者,我们尝试写入脚本的文本。打开二进制模式的文件,并利用静态 `QString::fromUtf8()` 方法把 `QFile::readAll()` 方法返回的 `QByteArray` 转换成 Unicode,而不是创建一个 `QTextStream`。一旦脚本就绪,让 `QWebPage` 的主框架对它进行评估,然后通知所有连接对象转换已成功。

`QWebFrame::evaluateJavaScript()` 方法返回一个 `QVariant`, 这个 `QVariant` 拥有最后执行的 JavaScript 表达式的值;我们忽略了它,因为我们已经选择了一个更为通用的方法来实现 JavaScript 和 C++ 之间的通信。既然已经为链接检查器设定了一个引用,链接检查器本身就是可以被 JavaScript 脚本读取的,那么脚本就可以调用任何链接检查器的公共槽,并获取它的属性值。在这个示例中,为使用 JavaScript 脚本而使用了 `addUrlAndName()` 槽。

```
void LinkFetcher::addUrlAndName(const QString &url,
                               const QString &name)
{
    if (url.startsWith("http://"))
        m_namesForUrls[url] = name;
    else
        m_namesForUrls[site + url] = name;
}
```

只要 JavaScript 脚本获得了详细的链接,它就会用链接的 URL 和名称调用这个槽,结果是填充 `m_namesForUrls` 槽。URL 可能是相对或是绝对的,我们把站点的路径预设为相对路径,站点会在建立时把这个相对路径传送给链接获取器。

图 1.11 显示了如何为一个网页添加应用程序的 `QObject`,在网页上下文中执行的 JavaScript 可以同时获取页面的 HTML 元素(通过文档对象模型)和所有页面访问的应用程序的 `QObject`。

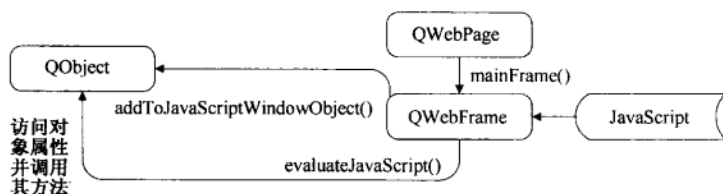


图 1.11 引入 JavaScript 来访问 HTML 元素和应用程序的 `QObject`

至此,已经介绍完了 `LinkFetcher` 类,但为了确保已经完全掌握了其原理,我们将给出 `fetch_article_links.js` 脚本(它是目前两个脚本中较短的),以帮助读者了解它如何工作、如何与链接获取器交互。

```

var links = document.getElementsByTagName("a");
for (var i = 0; i < links.length; ++i) {
    if (links[i].href.search("/articles/") != -1) {
        linkFetcher.addUrlAndName(links[i].href, links[i].innerHTML);
    }
}
  
```

JavaScript

JavaScript 的 `getElementsByTagName()` 方法用来提取所有的 `a` (anchor, 锚点) 标志——那些拥有链接、自身处于 `href` 属性中、文本在 `<a>` 和 `` 之间的标志,它拥有 `innerHTML` 属性。

需要特别注意的是最后一个声明,这里参考了 C++ 的链接获取器对象,用 `linkFetcher` 调用了 `LinkFetcher::addUrlAndName()`。

最初的 `fetch_issue_links.js` 脚本与 `fetch_article_links.js` 非常相似,但前者有点长(大约 20 行左右),它从显示在网站主页的组合框读取主题的链接。

笔者写成这本书大约花了 9 个月时间,NYRB 网站已经发生了巨大变化,主页和最初的 `fetch_issue_links.js` 脚本读取主题列表的主题组合框现在都已经不存在了。其他任何页面也都没有以任何方式提供类似的组合框或是主题列表。然而,网站管理员指出,网站对于每一年的主题列表都有单独的页面显示。根据这些信息,我们很容易地编写了一个新的 `fetch_issue_links.js` 脚本(大约 40 行左右)并把可执行的新脚本放在同样的目录下。我们设计了应用程序,它优先使用在同一个目录下发现的脚本,而不是使用嵌入在其资源中的脚本,把这些脚本作为可执行文件,它甚至不需要任何重编译,应用程序会自动使用新的 `fetch_issue_links.js` 脚本。

新的 `fetch_issue_links.js` 脚本必须读取一个记录了当年所有主题的页面(我们已把它设定为读取过去 5 年的主题),它比原来的脚本运行速度慢,其原因在于,它为每一个页面执行了同步的 GET 请求^①。尽管如此,这个解决方法很容易实施,并且表现良好。但从长远来看,我们希望避免在 JavaScript 中执行同步下载,因为这样会妨碍 GUI 应用程序的时间循环。这时,理想的解决办法

① 新的脚本使用 `XMLHttpRequest` 对象,它以 David Flanagan 撰写的 *JavaScript: The Definitive Guide*, ISBN 9780596101992 一书为基础。

是重新设计应用程序,让它用 Qt 的网络类来执行 GET 请求,在保留 `fetch_article_links.js` 脚本的同时获取主题列表。我们把这个留做练习,相信会很容易实现。

在结束本小节关于使用 JavaScript 的介绍之前,讨论一下如何使用类似引入 JavaScript 来调试应用程序是非常值得的(它并不容易)! 解决这个问题较简单的办法是确保在评估每一脚本时返回一个值,并检索、测试这个值。在本例中,我们采取了多种实现方法,并为链接获取器的头文件增加了一个新槽:

```
void debug(const QString &value)
{ qDebug("%s", qPrintable(value)); }
```

在开发过程中,我们在 JavaScript 脚本内部调用了这个槽(对 `debug()` 的调用仍然存在于源代码中,但被注释掉了)。例如,我们把下面的代码放入 `fetch_article_links.js` 脚本中来实现循环(虽然没有在前面引用脚本时讲解):

```
linkFetcher.debug(links[i].href + " * " + links[i].innerHTML); || JavaScript
```

添加类似的 `debug()` 声明非常有用(Windows 用户必须添加 `CONFIG += console` 到 `.pro` 文件,这样才能看到调试结果)。

遗憾的是,如果脚本含有语法错误,使用 `debug()` 声明就会无效,脚本也完全不会执行。检查脚本语法错误的一种方法是把它的文件名作为命令行参数传递给 Qt 的 `qscript` 程序(在 Qt 的名为 `examples` 的目录中)。如果没有语法错误,`qscript` 会尝试运行脚本;否则,它会提供一个错误信息,显示出第一个错误所在行的行号。

本小节介绍的 Books Viewer 应用程序的 New York Review 使用了相当简单的 JavaScript 脚本(虽然新的 `fetch_issue_links.js` 更为复杂)。我们可以写出更为成熟的脚本,特别是能访问一个所下载页面的 DOM(Document Object Model)脚本。我们还可以使用 Qt 4.6 的 `QWebElement` 类,并通过网页的 DOM 来访问(甚至修改)它,这对那些仍然稳定或我们已经拥有控制权的网站非常有用。因此,在这种情况下,我们必须在简单和方便之间做出抉择,改变我们的 JavaScript 脚本,使其与网站的改变一致,这样很方便,将它与 Qt 的联网能力、非同步性还有网络类相比较,就能做出决定,Qt 的网络类能很容易地显示一个快速响应的用户界面,并允许我们用纯 C++ 进行网络编程。

1.2.3 把 Qt 窗口部件嵌入到网页

HTML 中所能用的窗口部件是非常有限的。对此已有许多解决办法,如使用像 Flash 一样的私有内容格式;使用与为 Internet Explorer 开发扩展一样的私有浏览器扩展或通过一个嵌入式的 Java 应用程序来解决此问题。这些方法都要求浏览器的支持,对于所有跨平台的内容来说,这是不现实的。使用私有格式或扩展的另一个劣势在于我们受限于它们所提供的功能。

另一个办法是嵌入式 Qt 窗口部件。这样做的好处在于,我们可以完全控制所嵌入窗口部件的行为和外观,也拥有最大的自由度,可以内建任何想要的功能。但其缺陷在于,浏览器必须支持嵌入式的 Qt 窗口部件。

在这一小节中,我们将介绍图 1.12 所示的 Matrix Quiz 页面。这个页面内嵌在浏览器窗口组件中,并用于 Matrix Quiz 应用程序(`matrixquiz`)。

网页显示了两个随机生成的 3×3 矩阵,用户在第三个矩阵中输入合适的值以使等式成立,第三个矩阵的所有值都初始化为 0。如果用户按下 New 按钮,将产生一对新的矩阵。如果用户按下 Submit 按钮,将比较第三个矩阵的输入值和所应当具有的值,所有错误的值将以红色高亮显示。例如,图 1.12 中填有 181 和 187 的两个表格框。光标所在框以相反样式显示(也就是说,黑底白边),并带有一个矩形焦点框,例如,填有 116 的表格框。

网页由 HTML 元素和标准的自定义 Qt 窗口部件组成。标准的 HTML 元素包括标题文本、“Name:”标签、大的“+”和“=”符号等。名称行编辑框、按钮和结果标签也都是 HTML 元素,但对于本示例,我们使用的是标准 Qt 窗口部件。这些矩阵窗口部件都是自定义窗口部件(简单的 `QTableWidget` 子类)——这些部件在纯 HTML 中都是不可能实现的。

在实现过程中,我们早期创建的浏览器窗口组件不支持嵌入式的 Qt 窗口部件。增加一个自定义的 `QWebPage` 子类就可以了,这个子类包括所有需要的支持,我们需要把它传递给浏览器窗口,而不是让浏览器窗口创建自身的标准的 `QWebPage`。

我们从 Matrix Quiz 应用程序的 `main()` 函数中提取了三小段程序,从它们来开始介绍这个程序的全部代码。

```
qrand(static_cast<uint>(time(0)));

QWebSettings *webSettings = QWebSettings::globalSettings();
webSettings->setAttribute(QWebSettings::AutoLoadImages, true);
webSettings->setAttribute(QWebSettings::JavascriptEnabled, true);
webSettings->setAttribute(QWebSettings::PluginsEnabled, true);

QString url = QUrl::fromLocalFile(AQP::applicationPathOf() +
    "/matrixquiz.html").toString();

BrowserWindow *browser = new BrowserWindow(url, new WebPage);
browser->showToolBar(false);
browser->enableActions(false);
```

我们把 Qt 的全局函数 `qrand()` 作为 Qt 的随机数发生器。如果没有这里的调用,调用 `rand()` 总是返回相同的随机数结果(因为,只要没有明确设定,种子的默认值都是 1)。我们想使用 Qt 的随机函数,但并不是所有的平台(例如,一些嵌入式系统)都支持它,但这在本例中无关紧要,这些随机函数也是线程安全的(表 1.2 所示为 Qt 的全局函数,包括 `qrand()`)。

表 1.2 Qt 的全局公用函数

函数/示例	说 明
<code>u = qAbs(n);</code>	返回 <code>n</code> 的绝对值(正值)
<code>X = qBound(min, n, max);</code>	如果 <code>min <= n <= max</code> ; 返回 <code>n</code> , 否则, 如果 <code>n < min</code> 返回 <code>min</code> ; 否则返回 <code>max</code>
<code>QDebug(" %d: %s", integer, printable(string));</code>	利用 <code>printf()</code> 语法向控制台打印 C++ POD 类型——不熟悉的 Qt 类型(Windows 下的 .pro 文件, 需要“ <code>CONFIG += console</code> ”)
<code>QDebug() << number << string << hash << stringlist << map << variant << object;</code>	为控制台打印 C++ POD 类型 and 所有 Qt QObject, 包括像 <code>QHash</code> 和 <code>QMap</code> ; 需要 <code>#include <QDebug></code> 之类的集合(Windows 下的 .pro 文件, 需要“ <code>CONFIG += console</code> ”)
<code>b = qFuzzyCompare(f, g);</code>	如果浮点数(在 Qt 4.6 中为 <code>QTransform</code>) <code>f</code> 和 <code>g</code> 被认为相等, 返回 <code>true</code>
<code>x = qMax(n, m);</code>	返回 <code>n</code> 和 <code>m</code> 中的较大值
<code>x = qMin(n, m);</code>	返回 <code>n</code> 和 <code>m</code> 中的较小值
<code>const char *s = printable(qstring);</code>	从一个适合于 <code>printf()</code> 或 <code>QDebug()</code> 的 <code>QString</code> 返回一个 <code>char *</code> (利用本地 8 字节编码)
<code>x = qRound(f);</code>	返回将 <code>f</code> 去尾取整得到的值, 类型为 <code>int</code>
<code>x = qRound64(f);</code>	返回将 <code>f</code> 去尾取整得到的值, 类型为 <code>qint64</code>
<code>x = qrand();</code>	在 0 到 <code>RAND_MAX</code> 中间返回一个伪随机数(在 <code><cstdlib></code> 中定义)。这个线程安全函数使用的默认种子为 1; 调用 <code>qrand()</code> 以设定不同的种子
<code>qrand(u);</code>	利用 <code>uint u</code> 作为伪随机数发生器的种子
<code>s = qVersion();</code>	返回一个定义了应用程序所使用 Qt 版本的 <code>Qt const char *</code> (例如, “4.6.2”)

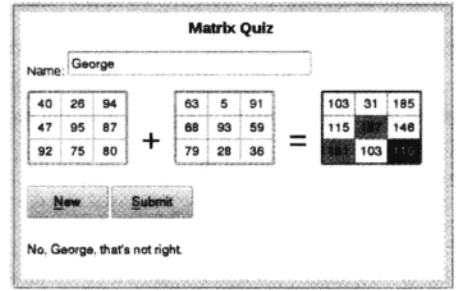


图 1.12 Matrix Quiz 网页

为了使用 JavaScript,就必须启用它,同样,为了使用嵌入式的窗口部件,就必须启用插件。

通常, QUrl 类用来为 Internet 上的网页创建 URL,但这里用它在本地文件系统中通过使用 file:// 方案为一个网页建立 URL。

当创建 BrowserWindow 示例时,不仅要把它传递给所要访问的网页(以字符串的方式)的 URL,还要把它传递给自定义的 WebPage 类。隐藏浏览器窗口的工具栏,关闭它的动作,现在,它就不能作为普通的浏览器使用了,而只能浏览我们定义的页面,并与之交互。

对于代码的其他部分,我们从较小的 WebPage 类开始介绍,它为嵌入式 Qt 窗口部件提供了支持。然后将介绍自定义的 MatrixWidget 类,最后介绍 matrixquiz. html 页面,以说明窗口部件是如何嵌入的,还将介绍用来提供一些页面功能所使用的 JavaScript。

WebPage 类是 QWebPage 的一个子类。构造函数(在此未介绍)把它的可选父参数传递给基类;它的函数体为空。子类仅重写了受保护的 createPlugin() 方法,但在了解它之前,先看一段从 matrixquiz. html 提取的代码,以说明其中的一个按钮是如何创建的。

```
<object type="application/x-qt-plugin" classid="QPushButton"
id="newButton" height="40" width="100">
Can't load QPushButton plugin!</object>
```

只要在 QWebPage 页面中碰到 type 属性为 application/x-qt-plugin 的 HTML <object> 标签,就会调用 createPlugin() 方法,如图 1.13 所示。

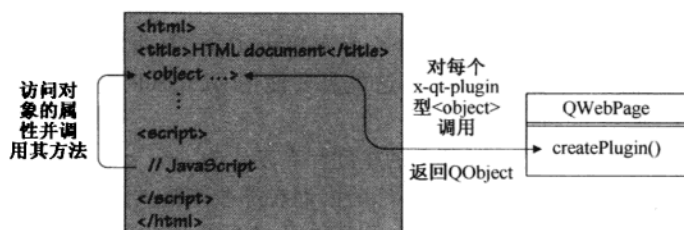


图 1.13 访问对象的属性并对调用对象的方法

```
QObject* WebPage::createPlugin(const QString &classId,
const QUrl&, const QStringList &parameterNames,
const QStringList &parameterValues)
{
    QWidget *widget = 0;
    if (classId == "MatrixWidget") {
        widget = new MatrixWidget(view());
        int index = parameterNames.indexOf("readonly");
        if (index > -1)
            static_cast<MatrixWidget*>(widget)->setReadOnly(
                static_cast<bool>(parameterValues[index].toInt()));
    }
    else {
        QUiLoader loader;
        widget = loader.createWidget(classId, view());
    }
    if (widget) {
        int index = parameterNames.indexOf("width");
        if (index > -1)
            widget->setMinimumWidth(parameterValues[index].toInt());
        index = parameterNames.indexOf("height");
        if (index > -1)
            widget->setMinimumHeight(parameterValues[index].toInt());
    }
}
```

```

    }
    return widget;
}

```

执行过程返回的默认结果是 0,在这种情况下,程序渲染的对象是 <object> 标志的文本(如果存在的话),而不是渲染预期目标。

classId 参数为 <object> 标记具体指定对象的类名称,将这个类名称作为 classId 属性。将另外一个 <object> 标志属性传递给两个平行的字符串列表,第一个获得属性名称,第二个获得相应的属性值。因此,对于我们前面看到的按钮 <object>,parameterNames 列表为[“id”,“height”,“width”],parameterValues 列表为[“newButton”,“40”,“100”]。

我们所用的方法从创建请求窗口部件开始,并把它作为 WebPage 的 QWebView 的子项。在 HTML 页面中,利用与 QWebPage 的相关 QWebView 渲染 createPlugin() 返回的所有非空窗口部件。

如果请求窗口部件类是 QWidget,当 parameterNames 列表含有一个“readonly”项时,可以创建一个合适的实例,并调用带有布尔值的 QWidget::readOnly()(在 HTML 中,已经把前两个矩阵设定为只读,第三个设定为可读写)。

如果请求的是其他类型的窗口部件,即任何标准 Qt 窗口部件,就可以用与创建 QWidget 相同的技术。但这会用到一个很长且难以维护的 if ... else 语句,以它来检查 classId。

值得庆幸的是,Qt 已经拥有了一个可以创建标准 Qt 窗口部件实例的类,标准 Qt 窗口部件都建立在它们的类名称 QUiLoader 之上。设计这个类的初衷是支持动态载入,并对 Qt Designer 的 .ui 文件进行渲染,但这里使用了它的 QUiLoader::createWidget() 方法来实例化所需的 <object> 窗口部件,并为其返回一个指针(值得注意的是,这里需要使用 QUiLoader 类,必须在应用程序的 .pro 文件中包含 CONFIG += uitools)。

一旦创建了窗口部件,如果把窗口部件的宽度和高度作为一个 <object> 属性,就要把它们设定为最小值。在我们的 C++ 代码中没有用 id 属性,而在那些能提供网页行为的 JavaScript 中使用了它,JavaScript 提供的行为要比 QWidget 和其他 Qt 类所提供的更为优秀。

我们希望自定义的 QWidget 类可以用在一个使用 JavaScript 进行编程的网页中。与使用 QScript 一样,在 JavaScript 中,可以把类的 Q_PROPERTY 作为 JavaScript 属性使用,一个类的公共槽可以作为 JavaScript 方法使用。以下是 QWidget 在其头文件中的定义。

```

class QWidget : public QTableWidgetItem
{
    Q_OBJECT
public:
    explicit QWidget(QWidget *parent=0);

public slots:
    void clearMatrix();
    void repopulateMatrix();
    QString valueAt(int row, int column) const
        { return item(row, column)->text(); }
    void setValueAt(int row, int column, const QString &value)
        { item(row, column)->setText(value); }
    void setHighlighted(int row, int column, bool highlight=true)
        { item(row, column)->setBackground(highlight ? Qt::red
                                                : Qt::white); }
    void setReadOnly(bool read_only);
};

```

我们定义了一些公共槽——在 JavaScript 中将可以对它们进行访问;但不需要创建任何自定义属性,因为基类的 rowCount 和 columnCount 属性可在 JavaScript 使用,它们也是我们唯一需要的属性。setHighlighted() 方法设定了一个表格项的背景——它用来高亮显示带有错误值的单元格。

```

const int ColumnWidth = 40;

MatrixWidget::MatrixWidget(QWidget *parent)
    : QTableWidgetItem(3, 3, parent)
{
    verticalHeader()->hide();
    horizontalHeader()->hide();
    for (int row = 0; row < rowCount(); ++row) {
        for (int column = 0; column < columnCount(); ++column) {
            QTableWidgetItem *item = new QTableWidgetItem("0");
            item->setTextAlignment(Qt::AlignCenter);
            setItem(row, column, item);
            if (row == 0)
                setColumnWidth(column, ColumnWidth);
        }
    }
}

```

使用构造函数创建一个 QTableWidgetItem, 它的行数和列宽度都是设定好的。初始化每一项, 让它居中显示文本 0。把水平和垂直的表头都隐藏掉, 这是窗口部件与一个标准的 QTableWidgetItem 的差异所在。

```

void MatrixWidget::setReadOnly(bool read_only)
{
    setEditTriggers(read_only ? QAbstractItemView::NoEditTriggers
                    : QAbstractItemView::AllEditTriggers);
    setFocusPolicy(read_only ? Qt::NoFocus : Qt::WheelFocus);
}

```

如果把窗口部件设定为只读, 那么关掉所有的编辑触发器。改变它的焦点策略, 让它不能接受键盘焦点; 这意味着, 当一个使用键盘的用户在只读矩阵窗口部件前按下 Tab 键, 焦点将跳过只读窗口部件直接定位到下一个可接受焦点的窗口部件。

在这个示例中, 焦点从可编辑的 name 行开始; 如果用户按下 Tab 键, 焦点将跳过中间的两个只读矩阵窗口部件, 定位到用户必须输入答案的可读写矩阵窗口部件。

```

void MatrixWidget::repopulateMatrix()
{
    for (int row = 0; row < rowCount(); ++row) {
        for (int column = 0; column < columnCount(); ++column)
            item(row, column)->setText(
                QString::number(qrand() % 100));
    }
}

```

这个方法用 0 ~ 99 之间的任意整数 (作为字符串) 填充窗口部件。clearMatrix() 方法 (在此未介绍) 在结构上与它非常相似, 唯一的不同在于, 它把每个单元格项的文本设定成 0, 把背景设置成白色。

我们已经看到了所有相关的 C++ 代码。matrixquiz.html 文件包含有定义 HTML 元素的 HTML 代码和一个 <object> 标志, 这个标志属于每一个 Qt 窗口部件和每一个 MatrixWidget。这些都和前面看到的相同, 以下是 MatrixWidget 的 <object> 标志, 它仅用来说明使用标准 Qt 窗口部件和自定义窗口部件没有区别:

```

<object type="application/x-qt-plugin" classid="MatrixWidget"
    id="leftMatrix" width="124" height="94" readonly="1">
Can't load MatrixWidget plugin!</object>

```

matrixquiz.html 文件用 JavaScript 提供了网页的行为。<script> 标志在文件尾端显示, 因为 JavaScript 需要获取前面创建的目标 (也就是那些 <object>)。在这段代码后, 定义了两个函数 (马上会介绍到)。

```
newButton.text = "&New";
submitButton.text = "&Submit";
resultLabel.text = "Enter the answer and click Submit";
repopulateMatrices();
newButton.clicked.connect(repopulateMatrices);
submitButton.clicked.connect(checkAnswer);
nameEdit.setFocus();
```

JavaScript

这里与 C++/Qt 的一个有趣的不同在于,在 JavaScript 和 QtScript 中,信号-槽连接是用以下这些语法中的一个建立的:

```
object.signalName.connect(functionName)
object.signalName.connect(otherObject.methodName)
```

JavaScript

我们将不介绍 `repopulateMatrices()` 函数,因为它所做的工作就是在每一个 `MatrixWidget` 处调用 `repopulateMatrix()`,但为了完整性,我们将介绍 `checkAnswer()` 函数。

```
function checkAnswer()
{
    var allCorrect = true;
    for (var row = 0; row < leftMatrix.rowCount; ++row) {
        for (var column = 0; column < leftMatrix.columnCount;
            ++column) {
            var highlight = false;
            if (Number(leftMatrix.valueAt(row, column)) +
                Number(rightMatrix.valueAt(row, column)) !=
                Number(answerMatrix.valueAt(row, column))) {
                highlight = true;
                allCorrect = false;
            }
            answerMatrix.setHighlighted(row, column, highlight);
        }
    }
    name = nameEdit.text == "" ? "mystery person" : nameEdit.text;
    if (allCorrect)
        resultLabel.text = "Yes, " + name + ", that's right!";
    else
        resultLabel.text = "No, " + name + ", that's not right.";
}
```

JavaScript

这个函数遍历三个矩阵的所有元素。如果答案矩阵的值正确,清除它的高亮显示效果(也就是说将它的背景设为白色),否则,我们将设定它为高亮效果(将其背景设定为红色)。如果有一个或更多的错误值,将 `allCorrect` 设定为 `false`。

最后,从 `nameEdit QLineEdit` 中检索用户的名字,根据用户是否正确回答问题,为 `resultLabel QLabel` 设定合适的文本。

至此,我们已经完成了 `Matrix Quiz` 网页和用来为网页提供外观和行为的 C++/Qt、JavaScript 的介绍。

把 Qt 窗口部件嵌入到网页,为增强网页功能提供了一种强大且成熟的方法,但它要求用户拥有一个支持 Qt 的网络浏览器,或使用一个支持 Qt 网络浏览窗口部件的应用程序,就像本章开发的 `browserwindow` 一样。有许多可以创建混合桌面/Internet 应用程序的方法,读者可以进行各种尝试。Qt 4.4 改进了这些方法,引入了支持 XQueries 和 XPath 语言的 `QtXmlPatterns` 模块。Qt 4.6 进一步改进了这些方法,引入了 `QGraphicsWebView` 类(`QWebView` 窗口部件的 `QGraphicsItem` 版本,为了在 `QGraphicsView` 出现而进行了优化),为了读取编辑一个 `QWebFrame` 的 DOM 元素和 `QWebElement` 类提供了一个漂亮的类似 jQuery 的 API。在特定情况下,Qt 允许我们选择任意正确的方法,这意味着,在我们编写的应用程序中,不必为了提供 Internet 功能而牺牲可用性、功能或与操作系统风格统一的界面外观。

第2章 声音和视频

- QSound 和 QMovie 的使用
- Phonon 多媒体框架

一直以来,Qt 4 能够读取 .wav 声音文件和带有动画的 .gif 和 .mng 文件。Qt 4.4 引入了 Phonon 多媒体库,Phonon 可以播放声音文件(如音乐)和视频文件(如电影),所以它的加入极大地扩展了 Qt 的多媒体处理能力。

短促的声音可以为用户提供有益的听觉提示信息,但它不能应付所有的情况,因为有些用户有听力障碍,有些用户将设备调到了静音或关闭了声音,或者根本没有声音输出设备。同样,动态图像也常被用来提示某项正在执行的操作,程序没有处于不响应的状态。它还被用来视觉化地表示当前选择了某一特定选项或者执行了某一特定操作。但它也可能使某些用户分散注意力甚至感到烦琐,所以要谨慎使用动态图像,用户要能够随时让它停止。

在本章的 2.1 节会首先介绍如何使用 QSound 和 QMovie 类向应用程序添加声音和动态图像。

多媒体应用程序旨在播放音乐或视频图像,与 QSound 和 QMovie 类相比,Phonon 多媒体框架具有更强大的功能和灵活性。2.2 节将介绍如何利用 Phonon 框架创建一个音乐和视频播放器。

Qt 的多媒体支持能够以相同的 API 运行于所有平台。然而,多媒体的播放能力通常取决于编解码器和第三方库,并且由于竞争的存在以及软件专利^①之间错综复杂的关系,通常不会安装它们。目前仍没有较好的方法来解决这个问题,所以声音和视频文件会存在不能跨平台播放的问题。未来最好的办法是利用诸如 Ogg(www.xiph.org/ogg)等开源媒体格式,尽管在写作本书之际,对开源媒体格式的支持在诸如 Windows 的那些专有平台上还不能像在 Linux 和 BSD 之类的开放平台上表现得那么好。

2.1 QSound 和 QMovie 的使用

播放较短的声音文件和动态图像(如 .gif 和 .mng 文件)最简单的办法是使用 QSound 和 QMovie 类。这两个类都提供了简单的 play() (QMovie 使用 start()) 和 stop() 方法。另外,QMovie 类能发射多种信号,包括 stateChanged() 信号(其参数为 QMovie::NotRunning, QMovie::Paused 或 QMovie::Running),frameChanged() 信号和 finished() 信号的情形也是一样的。

为了说明在上下文中如何使用这些类,我们将会介绍如图 2.1 所示的 Movie Jingle 应用程序。这个应用程序可以加载并播放动态图像,并可以使用 Qt 支持的任意基于像素的图片格式来保存图像的快照。另外,无论何时调用该应用程序,它都会播放一个很短的声音片段(一个“叮咚”声),它还提供了打开或者关闭该声音的开关功能。

我们可以通过建立一个 QAction 的子类,将“叮咚”声和动作连接起来,把这个子类命名为 JingleAction,这样就扩展了 QAction 的 API,把它提供了给“叮咚”声文件,并提供了一个静态布尔变

^① 有关软件专利的更多相关信息,可以参考 patentabsurdity.com 网站中的视频——“荒谬的专利”(Patent Absurdity);自由编程联盟(League for Programming Freedom)对专利相关的页面,可参见 progfree.org/Patents/patents.html; www.ifso.ie/documents/rms-2004-05-24.html 页面上 Richard Stallman 题为“软件专利的危害”(The Dangers of Software Patents)的演讲。

量来控制声音的播放和停止。建立 QAction 子类的主要优势在于它保证了无论用户如何调用这个动作(例如,通过快捷键、菜单选项或者工具条按钮),程序都会播放“叮咚”声(如果“静音”选项处于关闭状态)。

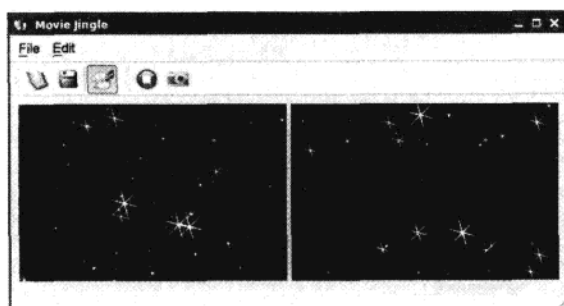


图 2.1 Movie Jingle 应用程序

JingleAction 类可以提供与 QAction 一样的构造函数,另外,还有接受“叮咚”声文件为参数的一对构造函数。在 jingleaction.hpp 的头文件中,实现了两个函数。

```
QString jingleFile() const { return m_jingleFile; }
static void setMute(bool mute) { s_mute = mute; }
```

私有变量 m_jingleFile 的类型为 QString,私有静态变量 s_mute 的类型为 bool,在 jingleaction.cpp 中初始化为 false。

构造函数不是简单地将“叮咚”声文件作为参数传递给基类,把自己变成空函数,而是以“叮咚”声文件作为参数在它的函数体内部调用一个方法,以下是来自 jingleaction.cpp 的例子:

```
JingleAction::JingleAction(const QString &jingleFile,
                           const QString &text, QObject *parent)
    : QAction(text, parent)
{
    setJingleFile(jingleFile);
}
```

在下面的代码中可以看到,程序中实现了其他两个自定义方法,一个是 setJingleFile(),另一个是个私有槽 play()。

```
void JingleAction::setJingleFile(const QString &jingleFile)
{
    if (!m_jingleFile.isEmpty())
        disconnect(this, SIGNAL(triggered(bool)), this, SLOT(play()));
    m_jingleFile = jingleFile;
    if (!m_jingleFile.isEmpty())
        connect(this, SIGNAL(triggered(bool)), this, SLOT(play()));
}
```

严格地说,断开和重新连接“叮咚”声动作的 triggered() 信号不是必须要有的,我们更倾向于在不必要的时候尽量避免发射信号。这里没有提供 clearJingle() 方法,因为调用 setJingleFile(QString()) 就已经足够了。

```
void JingleAction::play()
{
    if (!s_mute && !m_jingleFile.isEmpty())
        QSound::play(m_jingleFile);
}
```

当存在“叮咚”声文件并且静音选项处于关闭时,我们就可以播放声音文件,最好让声音的持续时间非常短(最多一秒钟),这样可以避免出现声音的持续时间超过执行请求动作持续时间的情况。

我们已经看到了执行 Jingle Action 过程的主要部分,下面利用部分代码片段演示如何使用已定义的“叮咚”声动作,如何使用 QMovie 类,这些代码是从 Movie Jingle 应用程序主窗口类提取的。让我们从程序主窗口类头文件中定义的两个枚举变量开始。

```
enum ReloadMode {DontReload, Reload};
enum MovieState {NoMovie, Stopped, Playing};
```

我们将看到上面这些枚举变量是如何在主窗口方法中使用的,下面是主窗口的构造函数:

```
const int StatusTimeout = AQP::MSecPerSecond * 5;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), movieState(NoMovie)
{
    movie = new QMovie(this);

    createActions();
    createMenusAndToolBar();
    createWidgets();
    createLayout();
    createConnections();

    AQP::accelerateMenu(menuBar());
    updateUi();
    statusBar()->showMessage(tr("Open a Movie file to start..."),
                             StatusTimeout);
    setWindowTitle(QApplication::applicationName());
}
```

构造函数以创建一个私有的叫做 movie 的 QMovie 对象开始,它被用来加载和播放一些动态图像。鉴于 createMenusAndToolBar()、createWidgets()、createLayouts() 和 createConnections() 都是标准的 C++/Qt 方法,这里不再赘述。当我们查看该应用程序的方法时会发现,这里建立的动作和它们所连接的槽的对应关系很明显,例如,打开文件的动作连接到了一个 fileOpen() 槽,其他亦然。

当然,我们还会从 createActions() 中抽取部分代码,来说明它与其他程序中所用的类似方法的差异。

一旦窗口部件被创建,相应的连接关系也确定后,程序会自动为所有的菜单和菜单项加上键盘加速键,并完成用户界面的初始化以供用户和程序进行交互。

虽然这里没有展示界面布局代码,但是我们能看到主窗口有两个并列的标签,左边用来显示影片内容,右边显示影片快照。

```
void MainWindow::createActions()
{
    jinglePath = AQP::applicationPathOf("jingles");
    imagePath = AQP::applicationPathOf("images");

    fileOpenAction = new JingleAction(
        jinglePath + "/fileopen.wav",
        QIcon(imagePath + "/fileopen.png"), tr("Open..."), this);
    fileOpenAction->setShortcuts(QKeySequence::Open);
    ...
}
```

这里仅演示第一个动作的建立过程,其他动作也是遵循相同的模式。但静音动作是个例外,它被设定为复选选项,初始值为未选择。

首先,要确定存放声音和图像的目录位置。文件路径在其他方法中一般被设置为私有实体变量,这里使用 UNIX 风格的路径分隔符以保证程序可以在所有平台上运行。

截至到本书写作时,QSound 类还不支持资源文件,因此“叮咚”声文件不能作为 Qt 资源,它们必须放在文件系统中。因此,要把应用程序的“叮咚”声保存在文件系统中,而不是编译进一个资

源集文件中,为保证一致性,也要对图像进行相同操作(前面探讨过 `AQP::applicationPathOf()`)。基于这样的认识,我们必须确认程序在运行时能够找到这些声音和图像文件。

下面来看程序所提供方法的具体实现,这些方法支持打开,播放和停止某一个影片文件,截取并保存影片快照,开启和关闭静音状态。

```
void MainWindow::fileOpen()
{
    QString fileFormats = AQP::filenameFilter(tr("Movies"),
        QMovie::supportedFormats());
    QString path(movie && !movie->fileName().isEmpty()
        ? QFileInfo(movie->fileName()).absolutePath() : ".");
    QString filename = QFileDialog::getOpenFileName(this,
        tr("%1 - Choose a Movie File"),
        .arg(QApplication::applicationName()), path, fileFormats);
    if (filename.isEmpty())
        return;

    movie->setFileName(filename);
    statusBar()->showMessage(tr("Loaded %1").arg(filename),
        StatusTimeout);
    movieState = Stopped;
    startOrStop(DontReload);
}
```

`QMovie::supportedFormats()` 静态方法返回一个文件后缀名的 `QList < QByteArray >`, 比如 [“gif”, “mng”]。`aqp. {hpp, cpp}` 模块的 `AQP::filenameFilter()` 函数基于这个列表返回一个 `QString`, 比如 “Movies(*.gif *.mng)”, 将这个字符串传递给 `QFileDialog::getOpenFileName()` 函数可以过滤要打开文件的类型。

如果用户选择了一个文件, 程序把影片的后缀名赋予这个文件并更新状态条。然后设定影片状态, 调用 `startOrStop()` 槽以立即开始播放影片。`startOrStop()` 方法的参数的默认值为 `Reload`, 但在这里我们设定它的值为 `DontReload`, 因为这时已经载入了影片。

```
void MainWindow::startOrStop(ReloadMode reloadMode)
{
    if (movieState == Stopped) {
        if (reloadMode == Reload)
            movie->setFileName(movie->fileName());
        movie->start();
        movieState = Playing;
    }
    else {
        movie->stop();
        movieState = Stopped;
    }
    updateUi();
}
```

如果要重新播放已经载入并播放过的影片, 就必须调用 `QMovie::setFileName()`, 新载入的影片就不需要这样做。影片处于开始播放还是停止播放状态取决于 `movieState` 的值, 通过 `updateUi()` 方法可以更新用户界面上当前影片的状态。

```
void MainWindow::updateUi()
{
    if (movieState == Playing) {
        startOrStopAction->setText(tr("&Stop"));
        startOrStopAction->setIcon(QIcon(imagePath +
            "/editstop.png"));
        startOrStopAction->setJingleFile(jinglePath +
            "/editstop.wav");
    }
}
```

```

    }
    else {
        startOrStopAction->setText(tr("&Start"));
        startOrStopAction->setIcon(QIcon(imagePath +
            "/editstart.png"));
        startOrStopAction->setJingleFile(jinglePath +
            "/editstart.wav");
    }
    startOrStopAction->setEnabled(movieState != NoMovie);
    takeSnapshotAction->setEnabled(movieState != NoMovie);
}

```

在这个方法中,我们更新 startOrStopAction“叮咚”声动作的文本、图标和声音文件,把它置于开始或停止动作。如果影片文件不存在(如在程序启动时),那么开始和停止动作以及快照动作都处于不可用状态。

```

void MainWindow::takeSnapshot()
{
    snapshot = movie->currentPixmap();
    fileSaveAction->setEnabled(!snapshot.isNull());
    snapshotLabel->setPixmap(snapshot);
}

```

当已载入某个影片(无论是正在播放还是已经停止)时,用户就可以使用 `QMovie::currentPixmap()` 方法来截取当前帧的快照,当截取结果是非空像素图时,程序会调用保存动作,无论在什么情况下都用快照标签窗口部件来显示图片。

```

void MainWindow::fileSave()
{
    if (snapshot.isNull())
        return;
    QString fileFormats = AQP::filenameFilter(tr("Images"),
        QImageWriter::supportedImageFormats());
    QString filename = QFileDialog::getSaveFileName(this,
        tr("%1 - Save Snapshot"),
        .arg(QApplication::applicationName()),
        QFileInfo(movie->fileName()).absolutePath(), fileFormats);
    if (filename.isEmpty())
        return;
    if (!snapshot.save(filename))
        AQP::warning(this, tr("Error"),
            tr("Failed to save snapshot image"));
    else
        statusBar()->showMessage(tr("Saved %1").arg(filename),
            StatusTimeout);
}

```

如果当前需要保存一个快照,程序弹出一个合适的 `QFileDialog`,它有一个用来过滤 Qt 可以读取的图像格式的过滤器。如果用户选择了文件名,程序就尝试以这个给定的名字来保存快照,而把按照后缀名确定使用文件格式的工作交给 `QPixmap::save()` 来完成。如果保存失败,程序弹出信息对话框,否则程序将在状态栏上显示保存成功(在题为“避免使用便捷 Qt 静态 `QMessageBox` 函数”的阴影部分中,说明了在向用户传递信息时,使用类似 `AQP::warning()` 的函数而不使用 Qt 标准的静态 `QMessageBox` 函数的原因)。使用 AQP 函数的另一个好处在于它把程序的名称(使用 `QApplication::applicationName()`)放置在标题栏里,所以,我们只需要传递标题的其他数据(对比调用 `QFileDialog::getSaveFileName()` 和 `AQP::warning()` 可以看出差异)。

```

void MainWindow::muteJingles(bool mute)
{
    JingleAction::setMute(mute);
}

```

在主窗口构造函数中所创建动作中有一个是 `fileMuteJinglesAction`, 它是一个初始状态为未被选择的用于切换的动作。把动作加载到菜单和工具栏并连接到这个槽, 用户就可以控制声音的播放了。

`QSound` 和 `QMovie` 类对于向用户提供听觉和视觉线索是非常有用的, 但这两个类都不能提供复杂的多媒体功能。幸好, 下一节介绍的 `Phonon` 多媒体框架提供了更多的多媒体功能, 需要的时候我们可以使用它。

2.2 Phonon 多媒体框架

`Phonon` 多媒体库是由 KDE 的开发人员建立的, 其初衷是让 KDE 的多媒体应用程序更易于编写。`Phonon` 有两个窗口部件, 提供通用多媒体 API 的前端和为这些 API 提供实际多媒体服务的后端。`Qt` 的 `Phonon` 模块为它的大部分 API 提供了一个轻量级的 `Qt` 风格的封装。对于任何想使用这个模块的开发项目来说, 必须为 `.pro` 文件加上 `QT += phonon`。

`Phonon` 最重要的特征之一是它可以跨平台使用, 这得益于它所支持的众多后端平台。在 Linux 环境下, 它通常使用 `GStreamer` 库; 在 Mac OS X 环境下, 它使用 `QuickTime`; 在 Windows 下, 它使用 `Direct X` 和 `DirectShow` 库。它还可以使用其他的后端平台库, 例如, `VLC` 和 `MPlayer` 后端 (code.google.com/p/phonon-vlc-mplayer)。在构建 `Qt` 应用程序时, 必须安装特定的后端平台——在 qt.nokia.com/doc/phonon-overview.html 页面提供了安装过程的提示说明^①。

避免使用 Qt 便捷的 QMessageBox 静态函数

`QMessageBox` 类为弹出模态对话框提供了一些静态便捷函数, 这些函数包含我们所需要的信息和按钮。然而, Mac OS X 的用户却希望看到 sheet, 而不是对话框。一个 sheet 就是一个模态对话框, 它在父窗口的标题栏中从上到下滑动到视图中, 最终停留在窗口最上面的中间位置 (就好像是窗口的一部分, 不可移动), 而不是在窗口最上面的中间弹出的可以移动和缩放的独立对话框。

为保证应用程序能够正确地跨平台, 我们创建了信息窗口便捷函数, `AQP::information()`、`AQP::warning()` 和 `AQP::question()`。还创建了一个 `AQP::okToDelete()` 函数, 它能够弹出一个带有 `Delete` 和 `Do Not Delete` 按钮的窗口; 一个 `AQP::okToClearData()` 函数, 它可以弹出一个带有 `Save`、`Discard` 和 `Cancel` 按钮的窗口。这两个函数都返回一个 `bool` 值, 如果用户需要保存信息, `AQP::okToClearData()` 需要调用 `save` 方法。这里是 `AQP::warning()` 函数的代码, 其他的几个函数的情形是类似的。

```
void warning(QWidget *parent, const QString &title,
             const QString &text, const QString &detailedText)
{
    QScopedPointer <QMessageBox> messageBox(new QMessageBox(parent));
    if (parent)
        messageBox->setWindowModality(Qt::WindowModal);
    messageBox->setWindowTitle(QString("%1 - %2")
        .arg(QApplication::applicationName()).arg(title));
    messageBox->setText(text);
    if (! detailedText.isEmpty())
        messageBox->setInformativeText(detailedText);
}
```

① 在撰写这本书的时候, 使用商业编译器时, 只支持对 Windows 平台下 `DirectShow` 后端的构建。

```
messageBox->setIcon(QMessageBox::Warning);  
messageBox->addButton(QMessageBox::Ok);  
messageBox->exec();
```

信息窗口只需要在这个 `AQP::warning()` 函数持续的时间内出现,当阻塞的 `exec()` 的调用结束时,它就关闭了。我们把信息窗口的指针存放在 `QScopedPointer` 中(或者在 Qt 4.5 中使用 `QSharedPointer`),这能保证一旦指针处于作用域之外时会被删除掉,进而避免了内存泄漏的危险,也把我们必须显式地删除这个指针的工作中解放出来(参见“Qt 的智能指针”中的阴影部分)。

将窗口模式设定为 `Qt::WindowModal` 是非常有必要的,这能保证窗口在 Mac OS X 中以 sheet 的形式出现,但在其他平台下,窗口仍然是标准的模态对话框。

Qt 的智能指针

Qt 4.0 引入了 `QPointer` 保护指针,Qt 4.5 引入了 `QSharedPointer` 和 `QWeakPointer` 智能指针,Qt 4.6 引入了 `QScopedPointer` 智能指针。这些指针类型封装了简单指针,通常比简单指针消耗更多的内存,读取速度也更慢。尽管如此,这些智能指针非常有用并且更加便捷,有助于避免内存泄漏——它们值得使用。

在利用 Qt 编写程序的过程中,因为 Qt 的父子所有权继承关系,很少会用到这些智能指针,因为需要调用 `delete` 的情况很少。任何时候只要我们需要调用 `delete`,或者是需要将某个指针设定为 0 时,应该考虑使用一个智能指针。

最常用的并且具有多种功能的智能指针是 `QSharedPointer`,它的行为更像是将要被下一个 C++ 标准所采纳的 Boost 库中的 `std::shared_ptr` 类。遗憾的是,`QSharedPointer` 的 API 与 `std::shared_ptr` 不是完全相同的。特别是在把简单指针封装在智能指针中的时候,它们所对应的方法分别是 `QSharedPointer::data()` 和 `std::shared_ptr::get()`。`QSharedPointer` 很智能的原因在于它能像正常指针一样被复制(这样,就可能有两个或更多 `QSharedPointer` 指向同一个对象),但是,如果最后一个(或者是仅有的) `QSharedPointer` 溢出了作用域,程序会自动删除它所封装的简单指针。

如果我们不对一个不具备 Qt 父子对象关系并且是在堆上分配的对象使用智能指针,那么当它不再被需要时,我们就要负责把它删除。仅在使用了指针的程序代码的尾部放置一个 `delete` 声明是不够的,因为可能代码还没有运行到 `delete` 声明的时候,异常就已经产生并导致函数过早地返回,这样会导致内存泄漏。解决这个问题的方法之一是使用一个 `try...catch` 结构,把 `delete` 声明放置在 `catch` 程序块中。如果要这样做,就需要保证能够捕捉到所有可能发生的异常,这里不能使用一个捕捉所有异常的处理代码,因为我们不想在偶然的情况下毫无知觉地捕捉预期之外的异常,这样会隐藏程序中的 bug。

最好的处理方法是使用 RAII (Resource Acquisition Is Initialization, 资源获取即初始化),实际上,它是指在作用域(或共享)指针构造函数调用时自行创建指针。现在,我们不需要担心自己删除指针以及异常的发生而导致函数过早返回,因为不管由于什么原因,只要作用域内的指针(或者指向一个对象的最后一个共享指针)溢出了作用域(),程序就会删除指针指向的对象。

使用 Qt 的父子对象关系以及这些智能指针,意味着对于一些应用程序来说,可以完全不用调用 `delete`。

Phonon 框架实际上有三种类型的对象:媒体数据源、媒体节点和媒体设备。在表 2.1 中列出了代表这些对象的类。

表 2.1 主要的 Phonon 类

类	说 明
Phonon::AudioOutput	音频数据汇点的媒体节点,用来驱动声卡或耳机
Phonon::Effect	处理器媒体节点,可以传递音频流
Phonon::EffectWidget	用来控制效果处理器参数的窗口部件
Phonon::MediaNode	所有媒体节点类型的基类
Phonon::MediaObject	媒体节点,用来控制多媒体对象的回放
Phonon::MediaSource	向媒体对象源节点提供媒体数据的对象
Phonon::Path	从一个媒体源节点到媒体对象汇点节点的数据路径
Phonon::SeekSlider	用来及时显示和修正媒体对象回放位置的部件
Phonon::VideoPlayer	可以加载、播放视频媒体并在后台自动处理媒体节点和路径建立过程的窗口部件
Phonon::VideoWidget	播放视频媒体的窗口部件
Phonon::VolumeSlider	显示并修正媒体对象音量的窗口部件

媒体数据源用 Phonon::MediaSource 的对象进行表示,它们通过文件、URL 或者像 QIODevice 这样的能够从中取得媒体数据的设备类的形式加以给定,它们自己不是媒体节点,并且只有当被赋予一个媒体对象源节点时才可以被使用。

媒体节点具有三种变体:源节点(不要与媒体数据源相混淆)、处理器节点和汇点节点(sink node)。源节点由 Phonon::MediaObject 对象表示,提供媒体回放界面。Phonon::MediaObject 拥有一个当前的 Phonon::MediaSource 对象,还拥有其他媒体源对象组成的队列,在一个媒体播放完成后接着播放另外一个。

Phonon::MediaObject 的输出一定会通过一条或多条路径到达汇点。一条路径用 Phonon::Path 对象表示,而且必须包含一个源节点(Phonon::MediaObject)和一个类似 Phonon::AudioOutput 或者 Phonon::VideoWidget 这样的汇点节点。路径可能是直线贯通的,也可能包含能够提供特殊效果的中间处理器节点。

Phonon 模块不支持直接对媒体流中的数据进行操作,但它能提供一种间接的方法来操作音频流,即效果处理器。这些处理器属于 Phonon::Effect 类,它们能被加载到源和汇点之间的路径上,在此之间传输数据。可用的效果依赖于 Phonon 的后端,由 Phonon::BackendCapabilities::availableAudioEffects() 函数得到。效果大致包括放大(amplification),立体全景图(stereo panorama)中的媒体流定位,均衡补偿(equalizing)和重采样(resampling)。

我们创建一个新的 Phonon::Effect 实例,对其应用特效以实现想要的效果。假设我们已经取得了 Phonon::createPath() 方法返回的 Phonon::Path 指针,接下来就可以使用它调用 Phonon::Path::insertEffect(), 并把刚刚创建的那个 Phonon::Effect 实例传给它。

汇点节点是数据最终被传递到的物理输出设备,比如声卡、耳机或视频设备,由 Phonon::AudioOutput 媒体节点或者 Phonon::VideoPlayer 以及类似媒体节点的 Phonon::VideoWidget 窗口部件来表示。图 2.2 说明了它们之间的关系。

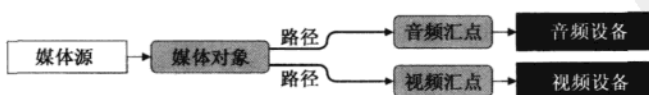


图 2.2 Phonon 架构

目前为止,我们已经从理论上体会了 Phonon 框架是怎样工作的,接下来的两个小节将以两个

示例说明如何在实践中使用 Phonon 框架,一个是音乐播放器,另一个是视频播放器(如果伴有音频的话,也会播放音频)。

2.2.1 播放音乐

这个小节中的 Play Music 示例(Playmusic,参见图 2.3)展示了如何创建一个音乐播放程序。这样的程序通常使用播放列表,一个数据库或者是文件系统来管理曲目,在本示例中我们选择使用文件系统。当程序启动时,界面的中央是一片空白区域,用来等待用户选择要播放的音乐目录。一旦用户选择了某个目录,程序就会遍历(iterate)目录中的所有音乐文件,递归(recurring)到所有子目录,并用“艺术家”、“专辑”和“曲目名称”以及每个曲目的持续时间等数据填充到一个 QTreeWidget 窗口部件。

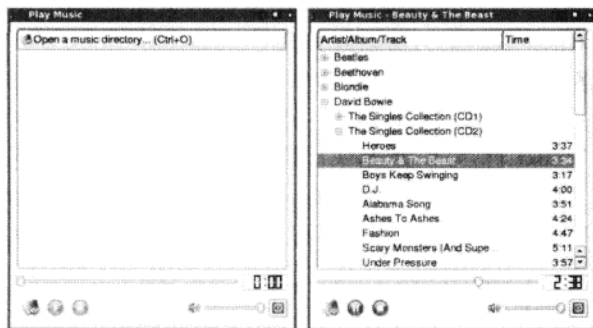


图 2.3 音乐播放程序的启动界面和带有音乐目录的界面

用户可以用鼠标或键盘跳转到某个特定的曲目,然后利用 Play/Pause 工具条的按钮或空格键(前提是要对空格键设定这个功能)对其进行播放或暂停操作。一旦曲目播放完成,程序会自动播放下一个曲目,除非用户点击了 Stop 按钮。

程序的数据存储在一个 QTreeWidget 中,对于艺术家和专辑,只存储它们的名称;但对于曲目来说,就需要保存它的名称和相应的文件名——都放在一个 TreeWidgetItem 里面,这是一个简单的 QTreeWidgetItem 子类,随后我们将讨论它(QTreeWidgetItem 是 Qt 的模型/视图便捷类之一,是一个自己提供所需模型的视图。模型/视图架构包括创建和使用自定义模型,这将在第 3 章到第 6 章中进行讨论)。

我们从主窗口的构造函数开始来了解这个程序是怎样建立的。

```
const int FilenameRole = Qt::UserRole;
const int OneSecond = AQP::MSecPerSecond;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), nextItem(0)
{
    playIcon = QIcon(":/play.png");
    pauseIcon = QIcon(":/pause.png");

    mediaObject = new Phonon::MediaObject(this);
    mediaObject->setTickInterval(OneSecond);
    audioOutput = new Phonon::AudioOutput(Phonon::MusicCategory,
                                          this);
    Phonon::createPath(mediaObject, audioOutput);

    createActions();
    createToolBar();
    createWidgets();
    createLayout();
}
```



```

createConnections();
setWindowTitle(QApplication::applicationName());
}

```

`nextItem` 是一个 `TreeWidgetItem`, 代表了下一个要播放的曲目, 当用户选择了一个曲目时, 它就会被设定, 当前的曲目播放到最后时, 它也会自动被设定。树部件项可以有多列数据, 每一列都可以使用一个 `QVariants` 的列表来保存用户数据。我们会进一步的看到, 可以利用代表曲目的树部件项的这个特性, 把曲目的文件名称保存在第一列的用户数据里面 (也就是 `column 0`) 使用 `FilenameRole` 常量作为项的用户数据的索引项。

这个私有的 `mediaObject` 对象用来播放用户选择的任意曲目, 声音数据被传递到私有音频输出汇点。必须根据它的用途来为音频输出指定声音的类别。这个类别用来确定音频输出的走向。例如, 在使用 VoIP (Voice over Internet Protocol, 网络语音协议) 的电话程序中, 声音将被传递到用户的耳机, 而音乐播放器的声音将被传递到声卡。

只要媒体对象和输出汇点存在, 就可以创建一个路径加入到其中, 如果我们不想为路径加入任何处理器实现特定的效果, 就可以不必保留一个对已建立的路径的引用。此外, 令媒体对象每秒 (1000 毫秒) 发出一个 `tick()` 信号以更新 LCD 数字窗口部件, 及时显示当前的播放位置。

`createActions()`、`createToolBar()` 或 `createLayout()` 方法没有任何特殊的地方, 此处不予讨论。但我们要看一看 `createWidgets()` 方法中的一个程序片段, 它说明了 Phonon 窗口部件是怎样建立的, 我们也要看看 `createConnections()`。

```

void MainWindow::createWidgets()
{
    seekSlider = new Phonon::SeekSlider(this);
    seekSlider->setToolTip(tr("Playback Position"));
    seekSlider->setMediaObject(mediaObject);
    volumeSlider = new Phonon::VolumeSlider(this);
    volumeSlider->setToolTip(tr("Volume Control"));
    volumeSlider->setAudioOutput(audioOutput);
    volumeSlider->setSizePolicy(QSizePolicy::Maximum,
                               QSizePolicy::Maximum);
    ...
}

```

播放位置滑块位于图 2.3 中所有滑块的最顶端, 它可以形象化地显示当前曲目已经播放的百分比, 用户还可以利用它来向前或向后跳动播放。这个滑块被附加到媒体目标上, 所以它可以反映当前的播放位置, 也可以改变播放位置。音量滑块 (它还包括一个静音按钮) 用来设定音量, 它附加于音频输出汇点上。

我们将分两部分介绍 `createConnections()` 方法。

```

void MainWindow::createConnections()
{
    connect(mediaObject, SIGNAL(tick(qint64)),
           this, SLOT(tick(qint64)));
    connect(mediaObject,
           SIGNAL(stateChanged(Phonon::State, Phonon::State)),
           this, SLOT(stateChanged(Phonon::State)));
    connect(mediaObject, SIGNAL(aboutToFinish()),
           this, SLOT(aboutToFinish()));
    connect(mediaObject,
           SIGNAL(currentSourceChanged(const Phonon::MediaSource&)),
           this, SLOT(currentSourceChanged()));
}

```

前面的 4 个连接把媒体目标和主窗口槽关联起来。`tick()` 连接用来更新 LCD 数字窗口部件, 这样它就能及时反映当前曲目的位置。`stateChanged()` 连接用来显示播放状态的改变, 例如, 它可以

用户在合适的时候启用或不启用这些控制窗口部件。实际上信号会同时发射出最新(当前)和历史(先前)两种状态。但在本程序中,我们仅关注最新状态。aboutToFinish()连接用来加载下一曲目到媒体对象序列以保证曲目播放的平滑过渡。当媒体对象源改变时,也就是说,当加载了新的曲目时,程序会释放出 currentSourceChanged()信号以更新用户界面的状态。

```
connect(setMusicDirectoryAction, SIGNAL(triggered()),
        this, SLOT(setMusicDirectory()));
connect(playOrPauseAction, SIGNAL(triggered()),
        this, SLOT(playOrPause()));
connect(stopAction, SIGNAL(triggered()), this, SLOT(stop()));
connect(treeWidget,
        SIGNAL(currentItemChanged(QTreeWidgetItem*,QTreeWidgetItem*)),
        this, SLOT(currentItemChanged(QTreeWidgetItem*)));
connect(treeWidget,
        SIGNAL(itemDoubleClicked(QTreeWidgetItem*, int)),
        this, SLOT(playTrack(QTreeWidgetItem*)));
connect(quitAction, SIGNAL(triggered()), this, SLOT(close()));
}
```

其余的连接用来提供用户界面的基本行为,如将音乐目录设定在最重要的位置供用户浏览;播放、暂停或停止正在播放的曲目,选择新的曲目或退出程序。

现在来回顾一下所有的槽,它们都和 Phonon 框架相关。让我们从使用最多的 setMusicDirectory()开始,以下的代码显示了如何为树结构填充数据,如何使用临时的媒体对象来获取某个曲目的信息。我们分三部分来说明此方法,先来看一下 setMusicDirectory()使用的私有帮助方法。

```
void MainWindow::setMusicDirectory()
{
    QString path = QFileDialog::getExistingDirectory(this,
        tr("Choose a Music Directory"),
        QDesktopServices::storageLocation(
            QDesktopServices::MusicLocation));
    if (path.isEmpty())
        return;
}
```

首先,用户需要选择音乐目录,其默认位置由静态的 QDesktopServices::storageLocation()方法提供。例如,在 Windows 环境下,返回目录的值可能是“% HOMEPATH% \My Documents\My Music”或者“% USERPROFILE% \My Documents\My Music”(环境变量会用实际的路径替换掉)。

```
QApplication::setOverrideCursor(QCursor(Qt::WaitCursor));
QSet<QString> validSuffixes = getSuffixes();
treeWidget->clear();
treeWidget->headerItem()->setIcon(0, QIcon());
treeWidget->setHeaderLabels(QStringList()
    << tr("Artist/Album/Track") << tr("Time"));
QHash<QString,TreeWidgetItem*> itemForArtist;
QHash<QString,TreeWidgetItem*> itemForArtistAlbum;
QDirIterator i(path, QDirIterator::Subdirectories);
while (i.hasNext()) {
    const QString filename = i.next();
    if (!QFileInfo(filename).isFile() ||
        !validSuffixes.contains(QFileInfo(filename).suffix()))
        continue;
    addTrack(filename, &itemForArtist, &itemForArtistAlbum);
}
```

首先得到的是一组文件后缀名,这些后缀名代表 Phonon 后端可以播放的音乐文件类型。我们建立两个哈希表(hash),一个用来为给定艺术家赋予合适的艺术家结构树项;另一个用来为给定艺术家和名称的专辑赋予合适的专辑结构树项(必须将艺术家和专辑名称结合起来,这样才能避免特殊情况下的冲突,如两个不同的艺术家都有同一个名字的专辑)。

为了在所有的音乐文件中遍历(traverse),首先要建立一个 QDirIterator 对象,它提供了一种在某个文件目录下遍历所有文件的简便方法,自动递归(recursing)到子目录,为所有文件、目录甚至是所遇到的文件系统对象命名。对于每个文件的名称(而不是目录、链接或其他文件系统对象)都会有一个合适的后缀名,可以通过调用 addTrack() 来为树结构窗口部件中的文件添加更多细节。

QFileInfo 类拥有类似的 isFile() 和 isDir() 方法来确定拥有某个名称的文件系统对象,它还有返回名称组成部分的方法,如 absolutePath()、fileName() 和 suffix()。其他的方法提供与文件访问权限有关的信息,如 isReadable() 和 isWritable(),关于文件状态方面的方法有 size()、created() 日期/时间和 lastModified() 日期/时间。

```
foreach (QTreeWidgetItem *item, itemForArtistAlbum)
    if (!item->childCount())
        delete item;
foreach (QTreeWidgetItem *item, itemForArtist)
    if (!item->childCount())
        delete item;
treeWidget->sortItems(0, Qt::AscendingOrder);
treeWidget->resizeColumnToContents(0);
stop();
QApplication::restoreOverrideCursor();
}
```

最后需要做一些清理工作。举个例子,当一个专辑没有曲目时就把它删除掉,同样,当一个艺术家没有专辑时也会被删除掉,然后程序会重新排序这些项并调整第一列的大小。当完全清除某个树结构并重新填充数据后,也需要停止之前播放的任何曲目。这意味着用户界面中前一阶段显示的所有曲目将不可用。

在默认的情况下,如果对一个树窗口部件的第一列来进行排序,这就会作用到所有级别的缩进(indentation),并对字符串进行大小写敏感的比较。我们希望项以不同的方式排序,因此,要用自定义 TreeWidgetItem 来为树结构填充数据,而不是使用继承自 QTreeWidgetItem 的对象。TreeWidgetItem 的构造函数(在此未介绍)简单地把参数传递给基类,它的函数体是空的。唯一添加的代码就是下面这个内联的 operator<() 成员函数的重新实现。

```
bool operator<(const QTreeWidgetItem &other) const
{
    QString left = data(0, FilenameRole).toString();
    QString right;
    if (!left.isEmpty())
        right = other.data(0, FilenameRole).toString();
    else {
        left = text(0);
        right = other.text(0);
    }
    return QString::compare(left, right, Qt::CaseInsensitive) < 0;
}
```

operator<() 方法必须为树结构窗口部件定义一个全局序列以便让其正常运行。乍看之下,代码并未提供这样的一个序列,其实不然。树窗口部件在其父项下独立地排序每一级别的缩进(indentation)(对于最高层项,可以认为在它的理论上的根项下执行这个操作)。在这样的情况下,“艺术家”被排列到一起;“专辑”被分别归类到其所属艺术家之下;“曲目”被归类到所属专辑之下。对于没有用户数据的艺术家或专辑,就对它们的文本进行比较;对于曲目来说,则比较它们的用户数据(文件名)。如果文件的命名方式是以曲目序号开始,就像 01-Space Oddity.ogg、02-Changes.ogg、03-Starman.ogg 那样,那么这种归类和排序的方法将非常有效,还保证了在所有层级都有一个全局序列。

要说明的其他两点和执行效率相关:第一,在比较艺术家或者是专辑时,仅仅需要验证其中一个项的用户数据是否为空,如果是的话,那么另一个项的用户数据一定为空;第二,在做比较时,不使用像 `left.toLower() < right.toLower()` 这样的方法,而是使用速度更快的 `QString::compare()` 函数(它返回一个 `int` 值),并把结果转换成合适的 `bool` 值。

现在我们看看 `setMusicDirectory()` 调用的两个帮助方法。

```
QSet<QString> MainWindow::getSuffixes()
{
    QStringList mimeTypeList;
    foreach (const QString &mimeType,
             Phonon::BackendCapabilities::availableMimeTypes())
        if (mimeType.startsWith("audio/"))
            mimeTypeList << mimeType;
    return AQP::suffixesForMimeTypes(mimeTypeList);
}
```

Phonon 框架会利用 MIME 类型来判定数据, `aqp. {hpp, cpp}` 模块的 `AQP::suffixesForMimeTypes()` 函数可接收一个 MIME 类型列表,并返回一组相应的文件后缀名。虽然这看上去很简洁,就像在“MIME 类型、文件后缀名和幻数”(Magic Number)的阴影部分中解释的那样,从底层来说,它并不是那么的简单易懂。

我们分两部分来介绍 `addTrack()` 方法。

```
void MainWindow::addTrack(const QString &filename,
                          QHash<QString, TreeWidgetItem*> *itemForArtist,
                          QHash<QString, TreeWidgetItem*> *itemForArtistAlbum)
{
    Phonon::MediaObject localMediaObject;
    Phonon::MediaSource source(filename);
    localMediaObject.setCurrentSource(source);
    if (!waitForMediaObjectToLoad(&localMediaObject, OneSecond))
        return;
    QString artist = localMediaObject.metaData(
        Phonon::ArtistMetaData).join("/").trimmed();
    QString album = localMediaObject.metaData(
        Phonon::AlbumMetaData).join("/").trimmed();
    QString artistAlbum = artist + "\t" + album;
    QString track = localMediaObject.metaData(
        Phonon::TitleMetaData).join("/").trimmed();
    qint64 msec = localMediaObject.totalTime();
}
```

局部媒体对象——`localMediaObject` 仅用来收集每个曲目的信息(元数据)。首先把媒体对象源设定为给定的文件名,媒体对象就会开始加载音乐文件,这仅仅会消耗很短(但不是零)的时间用来加载音乐文件以获取它的元数据(meta-data)。某些情况下,文件的元数据根本无法读取,比如,当文件损坏的时候。这时需要利用带有超时设定的本地事件循环(local event loop)来处理这种情况,在后面我们讨论 `waitForMediaObjectToLoad()` 方法时就会有所了解。

获取音乐文件的元数据,将每个 `QMediaObject::metaData()` 调用 `QStringList` 的返回值结合到一个单独的由“/”分割的 `QString`。`QMediaObject::totalTime()` 方法返回以毫秒为单位的曲目持续时间。我们没有获取的元数据的其中一项是 `Phonon::TracknumberMetaData`,它的值有时是空的,所以就像前面谈到的那样,我们倾向于通过文件名来点选曲目。

MIME 类型、文件后缀名和幻数

判定通过邮件接收和从 Internet 下载的文件或数据块(lump)类型有三种常用方法。第一

种,通过文件后缀名判别,如.png表示便携式网络图片(Portable Network Graphics, PNG)文件;第二种,使用幻数(magic number)——通常在每个文件的起始处都会有长度至少为一个字节的序列,例如, PNG 文件就是以 0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A 这 8 个字节开始的。

遗憾的是,以上两种方法都存在一个共同的问题:目前还没有官方标准。因此,有可能存在这样的情况:两个或更多的文件后缀名或幻数指向的是类型完全不同的文件或数据。这种导致重选的情况在实践中是会发生的,特别是对于文件后缀名来说。

第三种方法是使用一个 MIME 类型,例如, image/png 指的是 PNG 数据。MIME 类型是经过 IANA(Internet Assigned Numbers Authority, 因特网编号管理局)标准化的,这是它的优势所在,每一个 MIME 类型都与它所关联的文件是一一对应的。

Phonon 模块(顺便提及一下,Qt 的剪贴板和拖曳系统)使用 MIME 类型来判定它能够处理的文件或数据的类型,这种方法对于判定通过邮件接收或通过网络下载的数据非常有效。然而,遗憾的是,使用 MIME 类型来处理文件不是很方便,因为我们必须以某种方式将 MIME 类型映射到文件后缀名或幻数,虽然它的类型是经过标准化的,但 MIME 类型和文件夹或者幻数之间没有官方一致认同的映射关系。Linux 系统下,纯文本文件/etc/mime.types 提供了一个比较适当的映射,但它通常是不完整的。

在本书中提供的 aqp.hpp 模块的示例中,一个名为 `QSet<QString> suffixesForMimeTypes(const QStringList &mimeTypeNames)` 的函数使用了 mime.types 文件,如果函数(以及内在数据)可用的话,它可以提供一个适当的从 MIME 类型到文件后缀名的映射。这个函数可以在所有平台运行,并且可以很轻松地通过添加 mime.types 文件或改编代码来扩展它的 MIME 类型。

注意:苹果公司已经开发出了判定数据类型(包括文件和数据块)的第四种方法:统一类型标志符(Uniform Type Identifier, UTI)。其目的在于避开那些困扰其他判定方法的问题,不过,在本书成文的时候 UTI 还没有得到广泛采用。

这里的代码有些欠缺严谨,没有考虑元数据不完整或缺失的情况。一种解决办法是为文件名设定“佚名艺术家”和“佚名专辑”选项,并将曲目设定为可数,当读取每一个曲目时,它逐渐增加。

```
TreeWidgetItem *artistItem = itemForArtist->value(artist);
if (!artistItem) {
    artistItem = new TreeWidgetItem(
        treeWidget->invisibleRootItem(),
        QStringList() << artist);
    itemForArtist->insert(artist, artistItem);
}
TreeWidgetItem *albumItem = itemForArtistAlbum->value(
    artistAlbum);
if (!albumItem) {
    albumItem = new TreeWidgetItem(artistItem,
        QStringList() << album);
    itemForArtistAlbum->insert(artistAlbum, albumItem);
}
TreeWidgetItem *trackItem = new TreeWidgetItem(albumItem,
    QStringList() << track
        << minutesSecondsAsStringForMsec(msec));
trackItem->setData(0, FilenameRole, filename);
trackItem->setTextAlignment(1, Qt::AlignVCenter|Qt::AlignRight);
}
```

每一个“曲目”都必须作为“专辑”的子项放置在树结构中,同样,“专辑”是“艺术家”的子项。所

以,从检索 itemForArtist 哈希表(hash)开始查找对应当前曲目所属艺术家的树结构。如果当前是这个艺术家的第一个专辑的第一首曲目,那么这个项不存在。QHash::value()会返回一个默认构建的 TreeWidgetItem 值,也就是一个空指针。这种情况下要把这个项构建这个树的理论上的根的子项,也就是一个顶级项。无论是创建还是检索这个项,在结束操作时都需要有一个对应这个曲目的艺术家的树部件项。如果创建了这个项,要把它添加到 itemForArtist 哈希表(hash)中,以便在需要的时候能够检索到它。

可以对“专辑”项使用同样的处理流程(因为它是当前艺术家第一个专辑的第一首曲目),只有当必须创建它时,还要把“专辑”项设定为“艺术家”项的子项。最后得到一个曲目专辑的树部件项。如果创建了这个项,还需要把它添加到 itemForArtistAlbum 哈希表中。

最后,创建“曲目”项,将它设定为“专辑”项目的子项,而这个“专辑”项已经是它的“艺术家”项目的子项。把曲目项的第一列设定为曲目的名称,把它的第二列设定为以分和秒表示的曲目长度。我们还把曲目第一列的用户数据设定为曲目的文件名。

为完整起见,且鉴于后面还要用到这个函数,在此列出 minutesSecondsAsStringForMSec() 的代码:

```
QString MainWindow::minutesSecondsAsStringForMSec(qint64 msec)
{
    int minutes;
    int seconds;
    AQP::hoursMinutesSecondsForMSec(msec, 0, &minutes, &seconds);
    return QString("%1:%2").arg(minutes, 2, 10, QChar(' '))
        .arg(seconds, 2, 10, QChar('0'));
}
```

AQP::hoursMinutesSecondsForMSec() 函数接受一些毫秒值作为实参,它会把指针所传入的那些毫秒值代表的时间转换为由时、分、秒三个整数表达的时间。当 QString::arg() 的第一个参数值为整数时,其他的可选参数则为字段位数、基数和填充字符。

addTrack() 方法使用 waitForMediaObjectToLoad() 方法填充所有音乐文件的元数据。实现这个操作的一个简单的方法是使用一个内部 while 循环以持续监测元数据是否就绪,它还包括一个用来跳出循环的计时器(如果等待的时间太长)。这种做法的缺陷是这样的忙碌——等待循环会消耗很多不必要的 CPU 周期,因此,我们采用了更为高效的方法,即本地事件循环(local event loop)。

```
bool MainWindow::waitForMediaObjectToLoad(
    Phonon::MediaObject *mediaObject, int timeoutMSec)
{
    QEventLoop eventLoop;
    QTimer timer;
    timer.setSingleShot(true);
    timer.setInterval(timeoutMSec);
    connect(&timer, SIGNAL(timeout()), &eventLoop, SLOT(quit()));
    connect(mediaObject,
        SIGNAL(stateChanged(Phonon::State, Phonon::State)),
        &eventLoop, SLOT(quit()));
    timer.start();
    eventLoop.exec();
    return mediaObject->state() == Phonon::StoppedState;
}
```

首先,创建一个事件循环和一个给定超时时间的单触发计时器。两个信号-槽连接都可以停止循环,一个是由于超时,另一个是由于状态发生了改变,当这些都设定好以后,启动计时器和事件循环,然后等待事件循环完成。

当加载媒体对象时,它拥有 `Phonon::LoadingState` 状态。一旦媒体对象的状态发生改变,事件循环就会跳出。如果加载成功,媒体对象的状态就会变成 `Phonon::StoppedState`,然后就可以提取媒体对象的元数据了,此时程序返回 `true` 值。假设程序由于文件损坏或其他无法确定的原因返回 `false` 值,时间循环会因“超时”信号停止,这种情况下调用函数会跳过文件,放弃把这个文件加入到树中。

现在,我们已经完成了“音乐目录浏览”,并用检索到的所有可作曲目的详细信息填充了树。用户可以在这个树中进行浏览,只要当前项发生了改变,程序就会调用 `currentItemChanged()` 槽。

```
void MainWindow::currentItemChanged(QTreeWidgetItem *item)
{
    if (!playOrPauseAction->isEnabled()) {
        QString filename = item->data(0, FilenameRole).toString();
        if (!filename.isEmpty())
            playOrPauseAction->setEnabled(true);
    }
}
```

如果 `playOrPauseAction` 动作被禁用并且当前项为曲目时,就需要启用 `playOrPauseAction`,用户就可以按下 Play/Pause 按钮,当用户按下了这个按钮,程序就会调用 `playOrPause()` 槽。

```
void MainWindow::playOrPause()
{
    switch (mediaObject->state()) {
        case Phonon::PlayingState:
            mediaObject->pause();
            playOrPauseAction->setIcon(playIcon);
            break;
        case Phonon::PausedState:
            mediaObject->play();
            playOrPauseAction->setIcon(pauseIcon);
            break;
        default:
            playTrack(treeWidget->currentItem());
            break;
    }
}
```

如果一个曲目正处于播放状态,Play/Pause 按钮的作用为暂停。当用户暂停了媒体对象,按钮的图标就会改变,告诉用户当前为 Play 按钮。反过来,如果曲目处于暂停状态,播放媒体对象,然后将按钮变成 Pause 按钮。

如果曲目不处于播放状态,也没有暂停,那么用户一定是按下了 Play 按钮来播放一个新的项。这时需要为当前项调用 `playTrack()` 槽,当用户双击了一个项时也会调用这个槽。

```
void MainWindow::playTrack(QTreeWidgetItem *item)
{
    Q_ASSERT(item);
    QString filename = item->data(0, FilenameRole).toString();
    if (filename.isEmpty())
        return;
    if (!QFile::exists(filename)) {
        AQP::warning(this, tr("Error"),
            tr("File %1 appears to have been moved or deleted")
                .arg(filename));
        return;
    }
    nextItem = item;
    mediaObject->clearQueue();
    mediaObject->setCurrentSource(filename);
    mediaObject->play();
}
```

代码从尝试获取曲目的文件名开始,如果当前项为“艺术家”或“专辑”(这时没有文件名),或者在读取音乐目录之后文件已经被移走或删除(虽然这种情况下程序仍然要弹出一个错误信息(参阅“MIME 类型、文件后缀名和幻数”的阴影部分,就可以理解为什么使用的是 `AQP::warning()` 而不是 `QMessageBox::warning()`),则什么都不做。把 `nextItem` 设为当前项,清除媒体源的对象序列,并把当前源指向曲目的文件名,然后开始播放。这时,程序会调用 `currentSourceChanged()`。

```
const QString ZeroTime(" 0:00");

void MainWindow::currentSourceChanged()
{
    if (nextItem) {
        playOrPauseAction->setIcon(pauseIcon);
        timeLcd->display(ZeroTime);
        setWindowTitle(tr("%1 - %2")
            .arg(QApplication::applicationName())
            .arg(nextItem->text(0)));
        treeWidget->setCurrentItem(nextItem);
        nextItem = 0;
    }
}
```

如果存在下一项(比如用户按下了 Play 来播放一个选定的曲目或双击了一个曲目,程序调用了 `currentSourceChanged()` 槽),程序会更新用户界面,把 LCD 设定为 0:00,并在标题栏显示曲目名称,接下来下一项被设定为 0。

在曲目播放过程中的任何时候用户都可以按下 Pause 来暂停播放,按下 Play 来恢复播放或按下 Stop 停止播放。程序会在每次媒体对象状态改变时调用 `stateChanged()` 槽;在时钟嘀嗒时(每一秒)程序都会调用 `tick()` 槽;如果用户允许曲目播放到结束,在即将结束播放时,会调用 `aboutToFinish()` 槽。下面我们来看如何实现这些槽。

```
void MainWindow::stop()
{
    nextItem = 0;
    mediaObject->stop();
    mediaObject->clearQueue();
    playOrPauseAction->setIcon(playIcon);
    timeLcd->display(ZeroTime);
    setWindowTitle(QApplication::applicationName());
}
```

如果用户在当前曲目即将播放完成时点击了 Stop,程序会清除下一项内容以防止它自动播放,并停止播放当前曲目。同时还会清除媒体对象序列,以防下一曲目已经进入序列——等到学习 `aboutToFinish()` 时,我们会看到如何建立曲目序列。然后,程序会更新用户界面以反映当前状态。

```
void MainWindow::tick(qint64 msec)
{
    timeLcd->display(minutesSecondsAsStringForMsec(msec));
}
```

每次时钟嘀嗒(本例中为“每一秒”),程序都会调用这个槽更新 LCD 数字窗口部件的显示内容来及时反映曲目的播放位置。

```
void MainWindow::stateChanged(Phonon::State newState)
{
    switch (newState) {
        case Phonon::ErrorState:
            AQP::warning(this, tr("Error"),
                mediaObject->errorString());
            playOrPauseAction->setEnabled(false);
    }
```

```

        stopAction->setEnabled(false);
        break;
    case Phonon::PlayingState:
        playOrPauseAction->setEnabled(true);
        playOrPauseAction->setIcon(pauseIcon);
        stopAction->setEnabled(true);
        break;
    case Phonon::PausedState:
        playOrPauseAction->setEnabled(true);
        playOrPauseAction->setIcon(playIcon);
        stopAction->setEnabled(true);
        break;
    case Phonon::StoppedState:
        playOrPauseAction->setEnabled(true);
        playOrPauseAction->setIcon(playIcon);
        stopAction->setEnabled(false);
        timeLcd->display(ZeroTime);
        break;
    default:
        playOrPauseAction->setEnabled(false);
        break;
}
}

```

无论任何时候,只要媒体对象状态改变,程序就更新用户界面以保证某个动作处于“可用”或“不可用”状态,当错误发生时,程序弹出信息对话框来显示问题所在。

```

void MainWindow::aboutToFinish()
{
    QTreeWidgetItem *item = nextItem ? nextItem :
                                   treeWidget->currentItem();
    if (!item)
        return;
    item = treeWidget->itemBelow(item);
    if (!item) // Current track is the last track in the tree
        return;
    QString filename = item->data(0, FilenameRole).toString();
    if (filename.isEmpty()) { // item is an Artist or an Album
        item = item->child(0);
        if (!item)
            return;
        else {
            filename = item->data(0, FilenameRole).toString();
            if (filename.isEmpty()) // item is an Album
                item = item->child(0);
            if (!item)
                return;
            filename = item->data(0, FilenameRole).toString();
            if (filename.isEmpty())
                return;
        }
    }
    nextItem = item;
    Phonon::MediaSource source(filename);
    mediaObject->enqueue(source);
}

```

如果当前曲目播放完成,并且用户没有按下过 Stop,调用 aboutToFinish() 槽的目的是让程序继续播放当前曲目的下一首曲目。代码必须说明四种情况。最简单的情况是:当前曲目后没有任何项,这意味着最后的曲目已经播放,程序不需要再做任何事情,返回即可。其他三种情况都是当前曲目后还存在另一项。如果下一项为“艺术家”,程序必须找到这个艺术家第一个专辑的第一首曲

目。同样,如果下一项为“专辑”,程序必须找到它的第一首曲目。最后一种情况是,下一项只能是曲目了。一旦程序找到了要播放的曲目,就会把下一项设定为这个曲目的树结构项,然后基于这个项的文件名将媒体源添加到媒体对象序列。

倘若用户没有点击 Stop,在当前曲目播放完成时,媒体对象将会把媒体源定位到下一序列曲目并发射出 `currentSourceChanged()` 信号。这个信号连接到主窗口的 `currentSourceChanged()` 槽,它用来更新用户界面以反映出“新的曲目已经开始播放”的情况。

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    nextItem = 0;
    mediaObject->stop();
    mediaObject->clearQueue();
    event->accept();
}
```

当用户终止程序时,停止正在播放的音乐是非常重要的,否则音乐仍然会播放,虽然程序的窗口是不可见的。

到目前为止,我们已经完成了 Play Music 程序的学习。很明显,我们可以对它进行两方面的改进。第一,让程序“记住”音乐目录(例如,使用 `QSettings`),并把它作为程序启动时的默认打开目录。第二,第一次填充树结构时仅读取目录,如果目录中存在大量(几百或几千)的曲目,这样做会极大地提升程序的启动速度;只有当用户确实展开一个分支结构时,才填充每个专辑的细节内容。这两个改进留做练习。

虽然程序已经具备了所有需要的功能,我们没有使用所有可用的 Phonon 的 API 来播放音乐。尽管如此,前文也已经涉及了所有最重要的方面。其余就是关注细节了。仅以一个细节为例,控制用户的等待时间,即队列中的一个曲目播放完成进而播放下一曲目的时间间隔。这由 `transition-Time` 属性控制,它的默认值为 0,也就是没有时间间隔。赋予它一个负值可以实现曲目间的“渐入渐出式切换”;一个正值可以实现给定毫秒数的“无声切换”。另外,用编程实现即时搜索特定播放位置(如果媒体目标的潜在源支持的话),然后把它连接到一些媒体对象提供的其他信号,如 `finished()`。下一小节会介绍这个连接。

2.2.2 播放视频

播放视频与播放音乐十分相似,至少原理上是相似的。主要的区别在于播放视频不仅仅需要创建媒体对象到音频输出之间的路径,还需要建立一个媒体对象到视频输出之间的路径。下面以图 2.4 中的 Play Video 示例(playvideo)为例说明如何实现视频的播放。

这个示例每次加载并播放一个单独的视频文件,而不是像前一小节的 Play Music 应用程序那样,用户可以打开一个目录让播放器依次播放多个文件。这样,我们不需要为视频文件建立一个播放序列,因为在某个特定的时间段内,程序只读取并播放一个视频文件。尽管如此,视频播放程序在结构上与 Play Music 是十分相似的。所以,这里仅关注与 Phonon 相关的内容和那些不同于 Play Music 的部分。让我们从主窗口的构造函数开始。



图 2.4 正在播放视频的视频播放器(模拟)

```

const int OneSecond = AQP::MSecPerSecond;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    playIcon = QIcon(":/play.png");
    pauseIcon = QIcon(":/pause.png");

    mediaObject = new Phonon::MediaObject(this);
    mediaObject->setTickInterval(OneSecond);
    videoWidget = new Phonon::VideoWidget(this);
    Phonon::createPath(mediaObject, videoWidget);
    audioOutput = new Phonon::AudioOutput(Phonon::VideoCategory,
                                         this);
    Phonon::createPath(mediaObject, audioOutput);

    createActions();
    createToolBar();
    createWidgets();
    createLayout();
    createConnections();

    setWindowTitle(QApplication::applicationName());
}

```

在这里,程序创建了一个媒体对象用来管理视频,创建了一个路径来把数据从视频输出到视频窗口部件(也就是视频输出硬件),同时,把音频输出到音频输出硬件。所建立的音频输出路径仅对那些包含有音轨的视频文件起作用,由于所实现的程序要播放任意格式的视频文件,所以它能够播放音轨(如果存在的话)。另一个需要注意的是,必须将视频的类型设定为 `Phonon::VideoCategory`,对于音频来说类型是 `Phonon::MusicCategory`。

这里将跳过 `createActions()` 和所有其他动作的创建方法(除 `createConnections()` 外),原因是本示例和上一个示例有许多相同之处,例如,这里的 `createWidgets()` 方法几乎与 `Play Music` 程序中的 `createWidgets()` 方法完全一样,我们利用它创建了 `Phonon::SeekSlider` 和 `Phonon::VolumeSlider`。然而,需要注意的是,我们不一定非要使用 `Phonon` 框架提供的窗口部件,也可以结合已有的窗口部件创建或独立创建自定义的“搜索定位”和“音量控制”窗口部件。

```

void MainWindow::createConnections()
{
    connect(mediaObject, SIGNAL(tick(qint64)),
           this, SLOT(tick(qint64)));
    connect(mediaObject,
           SIGNAL(stateChanged(Phonon::State, Phonon::State)),
           this, SLOT(stateChanged(Phonon::State)));
    connect(mediaObject, SIGNAL(finished()), this, SLOT(stop()));
    connect(fullScreenAction, SIGNAL(triggered()),
           videoWidget, SLOT(enterFullScreen()));
    connect(stopAction, SIGNAL(triggered()), this, SLOT(stop()));
    connect(playOrPauseAction, SIGNAL(triggered()),
           this, SLOT(playOrPause()));
    connect(chooseVideoAction, SIGNAL(triggered()),
           this, SLOT(chooseVideo()));
    connect(quitAction, SIGNAL(triggered()), this, SLOT(close()));
}

```

代码中前三个连接(connections)把媒体对象连接到主窗口槽。`tick()`连接用来更新显示当前视频播放时间的 LCD 数字窗口部件。`stateChanged()`连接用来显示播放状态的改变,`finished()`信号用来探测视频文件什么时候完成播放。

其他连接用来提供基本的程序行为:启动、恢复播放、暂停、停止、选择视频文件来播放或者退出程序。

工具条中的一个按钮的连接是把视频窗口部件转换到全屏模式,当这个事件发生时,只有视频可见,用户就不能再通过窗口部件控制程序。基于这样的考虑,程序必须提供一些方法让用户可以退出全屏模式。我们已经通过提供一个快捷键(Esc)并设定一个事件过滤器实现了这个功能,也就是用户按下快捷键或只需点击视频窗口部件就可以退出全屏模式。以下是从建立事件过滤器和快捷键所用的 `createActions()` 方法中抽取的几行代码:

```
videoWidget->installEventFilter(this);
(void) new QShortcut(QKeySequence("Escape"),
                    videoWidget, SLOT(exitFullScreen()));
```

以下是事件过滤器的代码:

```
bool MainWindow::eventFilter(QObject *target, QEvent *event)
{
    if (target == videoWidget &&
        event->type() == QEvent::MouseButtonPress &&
        videoWidget->isFullScreen()
        videoWidget->exitFullScreen();
    return QMainWindow::eventFilter(target, event);
}
```

这个事件过滤器保证用户可以通过简单的点击就可以退出全屏模式。

还可以通过子类化 `Phonon::VideoWidget` 和重写它的 `mousePressEvent()` 方法来实现同样的效果。把这里建立的事件过滤器作为窗口部件行为的一个小扩展是个不错的做法。如果需要使用窗口部件的多个实例或要对窗口部件的行为做更多的修正和扩展,使用子类是最好的办法——特别是大量使用事件过滤器会导致程序运行效率降低。

下面来看一下这些槽,它们虽然与 `Play Music` 程序中的槽同名,但却存在不同之处。先从最长的 `chooseVideo()` 槽开始,我们将分三部分对它进行说明。

```
void MainWindow::chooseVideo()
{
    QString filename = QFileDialog::getOpenFileName(this,
        tr("Choose Video"), QDesktopServices::storageLocation(
            QDesktopServices::MoviesLocation), getFileFormats());
    if (filename.isEmpty())
        return;
```

`chooseVideo()` 槽由提示用户选择一个视频文件开始,它会提供给用户一个默认的“影片”目录(如果用户计算机没有存放影片的目录,则定位到用户的主目录)。如果用户取消选择,那么不做任何事情就返回。

```
stop();
playOrPauseAction->setEnabled(false);
stopAction->setEnabled(false);
mediaObject->setCurrentSource(filename);
if (!mediaObject->hasVideo()) {
    QEventLoop eventLoop;
    QTimer timer;
    timer.setSingleShot(true);
    timer.setInterval(3 * OneSecond);
    connect(&timer, SIGNAL(timeout()), &eventLoop, SLOT(quit()));
    connect(mediaObject, SIGNAL(hasVideoChanged(bool)),
        &eventLoop, SLOT(quit()));
    timer.start();
    eventLoop.exec();
}
```

一旦用户选定了某个视频文件,当前播放的视频文件就会停止,相关的动作会处于“禁用”状态。使用 `Phonon::MediaObject::setCurrentSource()` 在加载视频文件时,会马上播放,但对于与这个视频

关联的音频来说,则会出现明显的延时。如果此刻媒体对象的视频数据不是立即可用,应用程序会启动一个带有超时设定的本地事件循环,这与 Play Music 应用程序一样。不同之处在于,只要视频流的情况发生改变,也就是说,视频数据变得可用或已经超时了(本示例设定了一个长一些的时限为 3 分钟)应用程序会立即停止本地事件循环。

```

    if (mediaObject->hasVideo()) {
        QString title(mediaObject->metaData(Phonon::TitleMetaData)
            .join("/").trimmed());
        if (title.isEmpty())
            title = QFileInfo(filename).baseName();
        setWindowTitle(tr("%1 - %2")
            .arg(QApplication::applicationName()).arg(title));
        mediaObject->play();
    }
    else {
        setWindowTitle(QApplication::applicationName());
        AQP::warning(this, tr("Error"),
            tr("Cannot play video from %1").arg(filename));
    }
}

```

一旦本地事件循环完成,或者视频数据可用,应用程序就会获取元数据并开始播放视频;如果超时,应用程序会通知用户遇到了问题。我们不会明确地启动这些相关动作——都在 `stateChanged()` 槽里完成,只要视频开始播放,状态立即就会改变。

```

QString MainWindow::getFileFormats()
{
    QStringList mimeTypeTypes;
    foreach (const QString &mimeType,
        Phonon::BackendCapabilities::availableMimeTypes())
        if (mimeType.startsWith("video/"))
            mimeTypeTypes << mimeType;
    return AQP::filenameFilter(tr("Video"), mimeTypeTypes);
}

```

至此,加载文件的方法已经介绍完毕。应用程序创建了一个 Phonon 后端支持的视频格式的 MIME 类型列表,然后利用 `aqp. {hpp,cpp}` 模块的 `AQP::filenameFilter()` 函数建立一个文件后缀名列表。关于 MIME 类型到文件后缀名的映射,请参阅“MIME 类型,文件后缀名和幻数”的阴影部分。

除了一个小策略 (workaround) 之外, `stateChanged()` 槽与 Play Music 应用程序中的使用方法相同,这里仅展示不同之处。

```

case Phonon::PlayingState:
    videoWidget->setAspectRatio(
        Phonon::VideoWidget::AspectRatioWidget);
    videoWidget->setAspectRatio(
        Phonon::VideoWidget::AspectRatioAuto);

    playOrPauseAction->setEnabled(true);
    playOrPauseAction->setIcon(pauseIcon);
    stopAction->setEnabled(true);
    break;

```

当第一次播放一个视频文件时,应用程序会把视频文件的长宽比例缩放到与视频窗口部件显示区域一样大小。到目前为止,在某些特定系统下播放某种特定类型的视频文件时,这不一定能很好地工作。所以,我们需要手动调用两个不同的 `Phonon::VideoWidget::setAspectRatio()` 方法,以保证在所有情况下视频会在保证其长宽比的情况下被缩放到适应播放器窗口大小的尺寸。

```

void MainWindow::playOrPause()
{
    switch (mediaObject->state()) {
        case Phonon::PlayingState:

```

```
mediaObject->pause();
playOrPauseAction->setIcon(playIcon);
break;
case Phonon::PausedState: // Fallthrough
case Phonon::StoppedState:
    mediaObject->play();
    playOrPauseAction->setIcon(pauseIcon);
    break;
default:
    break;
}
}
```

这里使用的 `playOrPause()` 槽比在 `Play Music` 应用程序中使用的要简单得多,因为 `stop()` 和 `tick()` 槽还有 `closeEvent()` 槽和上一示例中几乎完全一样,所以这里不再赘述。

到此,我们已经完成了视频播放应用程序示例的学习。比使用 `Phonon::VideoWidget` 窗口部件更简易的选择是使用 `Phonon::VideoPlayer`。后者使用起来更加便捷——不需要创建源、媒体对象、路径或是汇点,但与本例中使用的视频窗口部件相比,这样做会付出缺乏更好的控制方法的代价。

`Phonon` 模块还包括 `Phonon::MediaController` 类,可以用它实现一些多媒体额外的控制功能,如 CD 标题、DVD 的章节和 DVD 的视角。在作者撰写本章内容时,还没有支持这些特征的 `QtPhonon` 后端。

本章基本上已经阐述了 `Phonon` 模块的所有内容。到目前为止,这个模块还不支持截取多媒体文件(如声音或视频片段)或存储多媒体文件以备后续播放。也不支持对媒体流进行处理,如编辑或对多种输入源进行混合。当 `Qt` 的 `Phonon` 模块更加成熟时,这些缺陷将会得到弥补。

`Qt 4.6` 引入了一个新的底层(low-level)的多媒体模块:`QtMultimedia`,它可以像 `Phonon` 模块那样读取并播放音频和视频数据。但底层的接口使得它使用起来比 `Phonon` 模块烦琐。如利用它来播放一个音频文件。必须创建一个包含各种音频格式技术细节的 `QAudioFormat` 对象,例如,音频的格式、通道数、采样率和编解码器的 MIME 类型。然后把音频格式对象传递给 `QAudioOutput` 对象播放,同时在音频文件中以二进制编码模式打开一个 `QFile` 对象。相比之下,这些低级、烦琐的工作会在 `Phonon` 模块中自动完成,我们所要做的只是给出一个文件名。使用 `Phonon` 模块将是一个良好的开端,最好仅仅在需要对多媒体进行底层控制(`Phonon` 模块不具备的功能)的时候才使用 `QtMultimedia` 模块。



第3章 模型/视图表格模型

- Qt 的模型/视图架构
- 用于表格的 QStandardItemModel
- 创建自定义表格模型

Qt 4.0 版本的一大进步是为数据项引入了模型/视图架构。开发者可以用这个框架方便地将数据与表现层区分开——有些功能在先前的版本中实现起来不太方便。随着 Qt 4.x 系列的演进,更多的功能和特性被加入到了这个架构中,使得它比首次出现时更加强大、可用和可靠^①。

本章是介绍 Qt 的模型/视图四个架构不同方面内容的第一个。本章将介绍表格模型(table model),在后续的章节中将依次介绍树模型(tree model)、委托(delegate)和视图(view)。

每个视图都提供了一个默认的委托——可以使用自定义的委托来替换显示每一项,并为可编辑项提供一个合适的编辑器。对于 Qt 自带的一些视图,我们所关心的仅是 QComboBox、一些需要模型来提供数据的显示部件和一些能够利用 Qt 的模型/视图架构的强大和灵活性的视图类,如 QTableView 和 QTreeView。当然,在介绍视图的第6章中也会介绍自定义视图。

Qt 还支持列表模型(list model),但在这里并不准备明确地进行介绍,因为在效果上它们与只有一列的表格模型是相同的(然而在第6章中,在要创建一个自定义的列表模型视图时,会用到一个列表模型)。我们会在下一章介绍树模型。

本章的3.1节首先会非常简短地向读者介绍一下 Qt 的模型/视图架构。接着在3.2节,会创建一个能够从文件中载入和保存 QStandardItem 项的 QStandardItemModel 派生类。在3.3节,将创建一个自定义表格模型来替换3.2节中的 QStandardItemModel 派生类模型。在3.2节和3.3节,读者就能明白如何添加行[使用一个委托的内置编辑(inplace editing)功能]、删除行和编辑行(仍然使用委托的内置编辑功能)。与此同时,还会创建两个 QSortFilterProxyModel 派生类,一个用来过滤出用户感兴趣的行,另一个用来过滤掉重复行。现在,作为背景介绍,我们一起来总览一下 Qt 的模型/视图的架构。

3.1 Qt 的模型/视图架构

正如在本章和下一章中所看到的,模型用于存储数据项(data item)。Qt 提供了几个窗口部件用于显示存储在模型中的那些数据项。有几种纯粹的视图部件:QListView、QTableView、QColumnView(能够以列表的横向序列方式来显示树型层次结构的视图,从 Mac OS X 中借鉴而来)和 QTreeView。所有的这些视图都必须提供一个模型(无论是自定义的,还是 Qt 中已提供的)来与之配合。Qt 中仍然提供了一些便利窗口部件(称其“便利”是因为它们提供了自己的内置模型并能直接使用),如 QListWidget、QTableWidget 和 QTreeWidget。还有 QComboBox,它既是一个便利窗口部件也是一个视图部件,这就是说,我们既能直接使用它(因为它提供了内置的模型),也能把它当做一个模型的视图部件(在这种情况下,可以提供一个合适的模型给它)。视图部件将在第6章中进行介绍。

^① 事实上,新一代 Qt 模型/视图架构已在开发中——尽管什么时候或者是否成熟并成为 Qt 的一部分仍然存在疑问。请访问 labs.qt.nokia.com/page/Projects/Itemview/ItemviewsNG 以了解进展。

所有的标准视图都提供了一个默认的 `QStyledItemDelegate` 委托——该类显示视图中的各个项,并为可编辑的项提供一个合适的编辑器。自然地,我们能创建自己的委托,来控制视图中的各个项的显示和编辑。关于委托,将在第5章中进行介绍。

关于模型、视图、委托以及其背后的数据之间的关系,如图3.1所示。

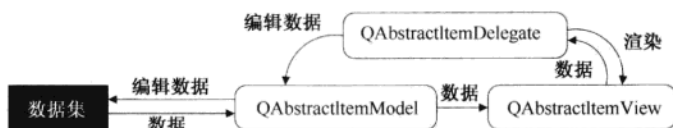


图 3.1 Qt 的模型/视图架构

在一些环境中,特别是对于那些数据集比较小(有数百或数千个项)的应用程序,假如它们使用的数据在任何时候仅仅显示在一个部件中,那么选择便利窗口部件(即内置模型视图部件)比较合适。在先前的章节中已经使用过了几个便利窗口部件,如在第1章中的 Matrix Quiz 示例用到的 `QTableWidget`,还有第2章中的 Play Music 示例中用到的 `QTreeWidget`。

尽管有统一的应用程序编程接口(API),仍然有两种不同类型的模型:有行列特性的表格模型,有父子关系的树模型,或许这是理解 Qt 中那些模型的最重要的关键点(实际上,列表模型与只有一列的表格模型相同)。在本章中将介绍表格模型,在第4章将介绍树模型。

除了考虑表格模型和树模型之外,还有三类模型可能用到。一种是 Qt 预定义的一些模型,如 `QStringListModel`、`QDirModel`,还有 `QDirModel` 的后继者 `QFileSystemModel`——这些模型可以直接使用并且只需做很少的工作。另一种是 `QStandardItemModel`,它是一个可被当做列表模型、表格模型或树模型来使用的通用模型。它提供了一个基于项的 API,以及那些便利窗口部件(即内置模型视图部件,如 `QTableWidget`)一样。对于那些能够直接把数据恰好地填充入列表、表格或树模型的各个数据项中,并能够直接使用或仅需做微小改动即可的情况,使用 `QStandardItemModel` 是理想的选择。最后一种是从 `QAbstractItemModel`(或 `QAbstractListModel`、`QAbstractTableModel`)派生而来的自定义模型。如果想达到最佳的性能,或者是该模型不符合基于项的特征,那就需要使用自定义模型了。一些 Qt 模型的层次结构如图3.2所示。

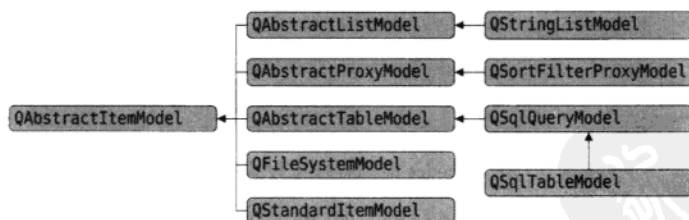


图 3.2 Qt 模型层次结构中的一些类

使用 `QStandardItemModel` 的话,那就不需要再创建自定义模型了,但它有两个潜在的缺点。首先,在加载大数据集时它明显地比自定义模型慢,其次,对于树模型来说,它提供的 API 不像自定义模型那样能有那么多的功能。尽管如此,大部分情况下可以先使用 `QStandardItemModel`,等到必要的时候再实现一个自定义模型来代替它。在题为“`QStandardItemModel` 和自定义模型比较”的阴影部分中有一个关于 `QStandardItemModel` 和自定义模型的简单对比。

3.2 用于表格的 QStandardItemModel

表格模型按照行和列进行工作,每一项的父项都是一个无效的 `QModelIndex` 对象。列表模型和表格模型并没有本质的不同——列表模型仅仅是只有一列的表格模型而已。

在本节中我们将了解如何创建一个自定义的 `QStandardItemModel` 派生类,它能够加载和保存自定义数据,并将每一项数据保存在 `QStandardItem` 对象中。在下一节中我们将使用一个自定义的 `QAbstractTableModel` 派生类来代替 `QStandardItemModel`,并且使用自定义的轻量级的对象保存数据。这两节中的应用程序具有完全相同的功能,然而在我们的测试机上,后者加载数据明显要快于前者。

两个示例都使用了自定义的 `ItemDelegate` 委托来处理显示和编辑,这部分将在第 5 章进行介绍。

两个 `Zipcodes` 应用程序(`zipcodes1` 和 `zipcodes2`)加载和保存内含邮编数据的二进制文件——包含有 `zipcode`(邮编)、`post office`(邮局)、`county`(县)和 `state`(州)这几个字段。这两个应用程序从外表看没有什么不同,且提供了相同的功能。其中之一的界面视图如图 3.3 所示。该程序提供了我们设想的所有标准功能,我们将介绍加载和保存、删除行、添加和编辑行这些功能。此外应用程序中还提供了以不同的方法选择、过滤数据的功能,因此当介绍这个应用程序的时候,在学习关于模型方面的内容的时候,我们还将了解如何创建 `QSortFilterProxyModel` 派生类和如何操作一个视图的选集模型(selection model)。

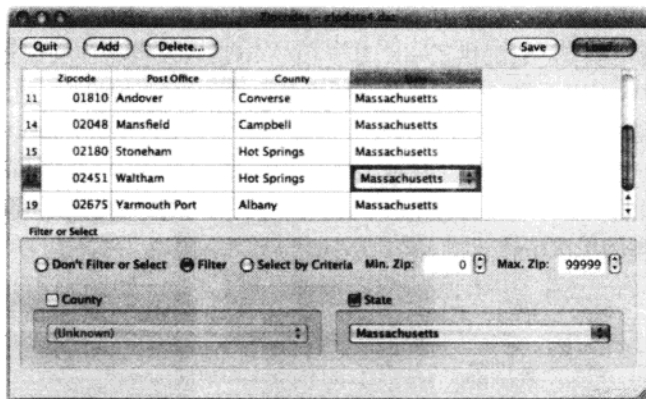


图 3.3 在 `Zipcodes` 应用程序中设置邮编所属的州

`zipcodes1` 应用程序使用了一个简单的 `QStandardItemModel` 派生类来加载、编辑和保存它的数据。`QTableView` 部件用来显示数据,它通过一个按照用户指定的条件进行过滤的 `QSortFilterProxyModel` 派生类来访问数据。组合框(combobox)用来过滤(或选择)行,它的数据由另一个 `QSortFilterProxyModel`(用于过滤掉重复数据)来填充。主窗体本身除了是对话框形式(使用按钮而非菜单)以外非常普通。一如既往,我们将关注相关细节(在这个示例中则是模型/视图方面),而忽略许多部件的创建和布局以及许多成员方法。

3.2.1 通过用户界面改变表格模型

首先从主窗体开始(它将给我们一个总体印象),然后将着眼于该应用程序所依赖的那些模型派生类。首先看一下程序的全局常量。


```
const int MinZipcode = 0;
const int MaxZipcode = 99999;
const int InvalidZipcode = MinZipcode - 1;
enum Column {Zipcode, PostOffice, County, State};
```

上面的常量可顾名思义。现在来看一下头文件中 `MainWindow` 类的定义,这里忽略了所有的私有方法和几乎所有的私有成员变量(即大部分窗口部件)。

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent=0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void load();
    void load(const QString &filename);
    bool save();
    void addZipcode();
    void deleteZipcode();
    void setDirty() { setWindowModified(true); }
    void updateUi();
    void radioButtonClicked();
    void selectionChanged();

private:
    ...
    QTableView *tableView;
    StandardTableModel *model;
    ProxyModel *proxyModel;
    bool loading;
};
```

槽函数 `load()` 和 `save()` 分别用来加载和保存应用程序的数据。这里并没有给出它们的代码,在后面介绍 `StandardTableModel` 派生类时,将列出它们调用的自定义 `StandardTableModel::load()` 和 `StandardTableModel::save()` 函数的代码。

槽函数 `addZipcode()` 和 `deleteZipcode()` 用来添加和删除数据行。这两个方法都将进行介绍。

大多数其他的私有槽函数是在用户与过滤和选择窗口部件的交互时被调用的。例如,如果用户在对应的组合框中选择一个特定的县。上面代码中的私有部分有若干个私有方法和大多数的窗口部件(它们大多并未显示在上面的代码段中,因为它们更多地与 GUI 相关,而非与模型/视图编程相关)。当用户选择过滤、选择动作或与条件选择相关窗口部件(比如设置邮编的最小值或者选择特定的州)进行交互时,槽函数 `updateUi()` 将被调用,它将相应的调用 `restoreFilters()` 函数来进行过滤或 `performSelection()` 函数来进行选择。其他的方法都是 C++/Qt GUI 开发中的标准代码。

`tableView` 这个成员变量被用做应用程序的视图, `model` 这个成员变量则被用于应用程序的模型。然而视图并不直接与模型进行数据传递,而是通过 `proxyModel` 这个用于过滤符合指定条件的行的成员变量来进行的。

```
const int StatusTimeout = AQP::MSecPerSecond * 10;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), loading(false)
{
    model = new StandardTableModel(this);
    proxyModel = new ProxyModel(this);
    proxyModel->setSourceModel(model);
```

```

createWidgets();
createComboBoxModels();
createLayout();
createConnections();

AQP::accelerateWidget(this);
setWindowTitle(tr("%1 (QStandardItemModel)[*]")
               .arg(QApplication::applicationName()));
statusBar()->showMessage(tr("Ready"), StatusTimeout);
}

```

构造函数的开头是创建视图用到的模型和代理模型(这些模型都将包含在后面介绍的派生类中)。构造函数的其余部分遵循了常见的模式。我们将跳过 `createLayout()` 方法而仅仅关注位于 `createWidgets()` 方法末尾的片段(在前面的章节中已经介绍了 `AQP::accelerateWidget()` 函数)。

```

void MainWindow::createWidgets()
{
    ...
    tableView = new QTableView;
    tableView->setModel(proxyModel);
    tableView->setItemDelegate(new ItemDelegate(this));
    tableView->verticalHeader()->setDefaultAlignment(
        Qt::AlignVCenter|Qt::AlignRight);
}

```

在上面的代码中可以看到,表格视图的模型是 `proxyModel`,而不是实际包含有数据的 `model`。自定义委托将在后面介绍。代码中设置了列表头(行号)的对齐方式为右对齐,以符合数字显示的习惯。

```

void MainWindow::createComboBoxModels()
{
    createComboBoxModel(countyComboBox, County);
    createComboBoxModel(stateComboBox, State);
}

void MainWindow::createComboBoxModel(QComboBox *comboBox, int column)
{
    delete comboBox->model();
    UniqueProxyModel *uniqueProxyModel = new UniqueProxyModel(column,
                                                                this);

    uniqueProxyModel->setSourceModel(model);
    uniqueProxyModel->sort(column, Qt::AscendingOrder);
    comboBox->setModel(uniqueProxyModel);
    comboBox->setModelColumn(column);
}

```

两个用于过滤和选择行的组合框需要从底层模型(即 `model` 成员变量)获取数据,尽管如此,在不同的邮编下相同的县和相同的州一再显示出来,如图 3.3 所示。因此,这两个组合框不是使用 `model`,而是使用自定义的代理模型来去除重复项。

在创建一个新的组合框模型前应先删除其内置的旧模型。这是因为每当加载新的数据文件时要调用 `createComboBoxModels()` 方法,我们想使用户能够在已加载的模型数据中进行数据过滤和选择。例如,可能加载了一个只包含 Connecticut 和 Delaware 两地邮编的文件,在这种情况下不希望用户能够过滤 Montana 州,因为没有符合条件的数据行。

自定义模型 `UniqueProxyModel` 与底层模型(成员变量 `model`)使用了同样的列,在创建代理(proxy)时必须明确要对哪一列进行唯一值过滤。设置代理的源模型(source model)为底层模型 `model`,设置组合框的模型为 `proxy`——组合框的下拉列表框内应显示通过 `QComboBox::setModelColumn()` 方法设置的指定列[唯一代理模型(unique proxy model)将在后面进行介绍]。

该应用程序使用了大概 20 个信号-槽连接,在 `createConnections()` 方法中设置。下面分成 4 组来介绍。

```
connect(model, SIGNAL(itemChanged(QStandardItem*)),
        this, SLOT(setDirty()));
connect(model, SIGNAL(rowsRemoved(const QModelIndex&,int,int)),
        this, SLOT(setDirty()));
connect(model, SIGNAL(modelReset()), this, SLOT(setDirty()));
```

上面这些信号都被连接到 `setDirty()` 槽函数(调用 `setWindowModified()` 方法)。它们需要确保我们知道界面上的数据是否有未保存的修改,并将其反映到标题栏中——在标题栏中多了一个“*”号或者在 Mac OS X 下在关闭按钮中包含一个点。

```
connect(countyGroupBox, SIGNAL(toggled(bool)),
        this, SLOT(updateUi()));
connect(countyComboBox,
        SIGNAL(currentIndexChanged(const QString&)),
        this, SLOT(updateUi()));
...
foreach (QRadioButton *radioButton, QList<QRadioButton*>()
        << dontFilterOrSelectRadioButton << filterRadioButton
        << selectByCriteriaRadioButton)
    connect(radioButton, SIGNAL(clicked()),
            this, SLOT(radioButtonClicked()));
```

我们需要知道用户是否选中(或取消选中)了代表县的群组框上的复选框,这样才能按照县进行过滤或选择(或取消过滤或选择)。如果群组框上的复选框处于选中状态并且用户变更了所选择的县,则程序必须相应地进行重新过滤或选择(对于代表州的群组框、组合框也有同样的信号连接——这里没有列出来,因为它们与“县”的相同)从邮编数字微调框(`spinbox`)的 `valueChanged()` 信号连接到 `updateUi()` 的槽函数的代码,同样没有在上面的代码中列出。

用户可以决定是否进行过滤或选择,这取决于哪一个单选框被选中了,单选框的连接用来确保我们进行正确的过滤或选择。

```
connect(tableView, SIGNAL(clicked(const QModelIndex&)),
        this, SLOT(selectionChanged()));
connect(tableView->selectionModel(),
        SIGNAL(currentChanged(const QModelIndex&,
                                const QModelIndex&)),
        this, SLOT(selectionChanged()));
connect(tableView->horizontalHeader(),
        SIGNAL(sectionClicked(int)),
        tableView, SLOT(sortByColumn(int)));
```

上面一组调用中的前两个连接调用,是在用户点击或操作特定的项时用来刷新用户界面的。第三个连接调用是用来提供排序功能的(关于排序的工作原理将在后面介绍)。

需要注意的是,视图与两个模型相关联——一个用来提供数据,另一个内部模型则用来追踪选集状态。选集模型可以通过 `QAbstractItemView::selectionModel()` 方法来获取,它的类型是 `QItemSelectionModel`(它是 `QObject` 的直接子类,而不是 `QAbstractItemModel` 子类)。

```
connect(loadButton, SIGNAL(clicked()), this, SLOT(load()));
connect(saveButton, SIGNAL(clicked()), this, SLOT(save()));
connect(addButton, SIGNAL(clicked()), this, SLOT(addZipcode()));
connect(deleteButton, SIGNAL(clicked()),
        this, SLOT(deleteZipcode()));
connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
```

上面的最后一组连接都比较常见。是用户触发的加载、保存数据,添加、删除行,退出应用程序时要响应的动作。

现在我们已经对程序的结构和信号-槽连接情况有了个总体的了解。大多数私有槽函数和私有方法都与用户界面相关,而与模型/视图编程关系不大,因此我们将跳过这部分与用户交互相关的槽函数和方法,而仅介绍所有的与模型/视图编程相关的内容。

```
void MainWindow::radioButtonClicked()
{
    if (dontFilterOrSelectRadioButton->isChecked()) {
        proxyModel->clearFilters();
        QItemSelectionModel *selectionModel =
            tableView->selectionModel();
        selectionModel->clearSelection();
    }
    else
        updateUi();
}
```

如果用户点击了“Don't Filter or Select”单选按钮,则清除过滤和选集状态——这样,所有的行都能显示出来。否则根据用户所选条件,调用 `updateUi()` 方法来进行选择或过滤。

```
void MainWindow::updateUi()
{
    if (loading || dontFilterOrSelectRadioButton->isChecked())
        return;
    if (filterRadioButton->isChecked())
        restoreFilters();
    else
        performSelection();
}
```

每当用户改变复选框、组合框或微调框的状态的时候, `updateUi()` 方法就会被调用。如果数据正在被载入或者用户选择了不进行过滤和选择,则直接返回。否则根据用户所选条件进行过滤或者选择。

`performSelection()` 方法内容比较多,这里把它分成两部分来进行介绍。

```
void MainWindow::performSelection()
{
    proxyModel->clearFilters();
    int minimumZipcode = minimumZipSpinBox->value();
    int maximumZipcode = maximumZipSpinBox->value();
    QString county = countyGroupBox->isChecked()
        ? countyComboBox->currentText() : QString();
    QString state = stateGroupBox->isChecked()
        ? stateComboBox->currentText() : QString();
}
```

首先清除过滤条件,不过滤任何行,然后获取用户所设邮编的最大值和最小值,以及用户所选择的县和州(如果没有选择,则设置为空字符串)。

```
QItemSelection selection;
int firstSelectedRow = -1;
for (int row = 0; row < proxyModel->rowCount(); ++row) {
    QModelIndex index = proxyModel->index(row, Zipcode);
    int zipcode = proxyModel->data(index).toInt();
    if (zipcode < minimumZipcode || zipcode > maximumZipcode)
        continue;
    if (!matchingColumn(county, row, County))
        continue;
    if (!matchingColumn(state, row, State))
        continue;
    if (firstSelectedRow == -1)
        firstSelectedRow = row;
    QItemSelection rowSelection(index, index);
    selection.merge(rowSelection, QItemSelectionModel::Select);
}
```

```

QItemSelectionModel *selectionModel = tableView->selectionModel();
selectionModel->clearSelection();
selectionModel->select(selection, QItemSelectionModel::Rows|
                      QItemSelectionModel::Select);
if (firstSelectedRow != -1)
    tableView->scrollTo(proxyModel->index(firstSelectedRow, 0));
}

```

选集对象必须一行一行地构造。首先声明一个空的 `QItemSelection` 对象,然后遍历代理模型中的每一行(以构造 `QItemSelection` 对象)(这意味着考虑到了底层模型的每一行,因为已经去除了代理模型的过滤条件)。当有一行符合用户所设置的条件时,则创建一个包含该行的 `QItemSelection` 对象,然后把它并入我们构建的打算包含所有选择行的 `QItemSelection` 对象中。

一旦处理完所有行,就清除现存的选集模型,然后选中符合用户所选条件的行(当然可能是空的)。最后,滚动到选中行的第一条。

```

bool MainWindow::matchingColumn(const QString &value, int row,
                                int column)
{
    if (value.isEmpty())
        return true;
    QModelIndex index = proxyModel->index(row, column);
    return value == proxyModel->data(index).toString();
}

```

这个辅助方法用于判断给定的值是否与指定行、列的数据相符。当给定的值与指定的行、列的值相同或者给定的值为空时,就返回 `true`。如果用户没有指定,比如未指定县,那么该方法对于任何县都将返回 `true`。

```

void MainWindow::restoreFilters()
{
    proxyModel->setMinimumZipcode(minimumZipSpinBox->value());
    proxyModel->setMaximumZipcode(maximumZipSpinBox->value());
    proxyModel->setCounty(countyGroupBox->isChecked()
        ? countyComboBox->currentText() : QString());
    proxyModel->setState(stateGroupBox->isChecked()
        ? stateComboBox->currentText() : QString());
    reportFilterEffect();
}

```

如果用户选中了 `Filter` 单选按钮,或者在 `Filter` 单选按钮处于选中状态时改变了任一个组合框或数字微调框,那么将调用 `restoreFilters()` 方法。它仅仅是使用自定义代理模型的对应方法来设置过滤条件以与用户界面中所示条件一致,并且这会使得视图更新自己的显示(以保持同步)。

```

void MainWindow::reportFilterEffect()
{
    if (loading)
        return;
    statusBar()->showMessage(tr("Filtered %L1 out of %Ln zipcode(s)",
        "", model->rowCount()).arg(proxyModel->rowCount()),
        StatusTimeout);
}

```

在用户点击了 `Filter` 单选按钮或者改变了过滤条件,就会调用 `reportFilterEffect()` 方法,从而在状态栏显示当前数据集中有多少行以及其中过滤掉多少行的信息。

为使数字在行数众多时更具可读性,这里使用诸如 `%L1`、`%L2`,而不是使用诸如 `%1`、`%2` 的形式,来提供国际化数字分隔符。例如,在美国每三位数字用逗号分隔。在本例中我们希望翻译器(translator)能够自动地以“... out of %Ln zipcode(s)”的形式来翻译合适的复数形式内容(比如“...out of one zipcode”或者“...out of %Ln zipcodes”);我们将在题为“使用三个参数形式的 `tr()` 函数”的阴影部分中讨论这个问题。

```

void MainWindow::addZipcode()
{
    dontFilterOrSelectRadioButton->click();
    QList<QStandardItem*> items;
    QStandardItem *zipItem = new QStandardItem;
    zipItem->setData(MinZipcode, Qt::EditRole);
    items << zipItem;
    for (int i = 0; i < model->columnCount() - 1; ++i)
        items << new QStandardItem(tr("Unknown"));
    model->appendRow(items);
    tableView->scrollToBottom();
    tableView->setFocus();
    QModelIndex index = proxyModel->index(proxyModel->rowCount() - 1,
                                           Zipcode);
    tableView->setCurrentIndex(index);
    tableView->edit(index);
}

```

如果用户选择添加新邮编,那么首先要关闭过滤和选择功能。关闭过滤非常重要,因为如果新添加的邮编不符合过滤条件,那它将立即被过滤掉,这样的话尽管它已经被添加到了数据中,用户却看不到,也无法进行编辑。

因为使用了简单的 `QStandardItemModel` 派生类以及将数据保存于 `QStandardItem` 对象,添加一个新邮编的操作只要适当地初始化一行的几个 `QStandardItem` 并把新行添加到模型(model)中即可。新数据被添加到底层模型(underlying model)中——视图使用的代理模型将会自动探测这种情况并随之适应。这里需要指出的一个不那么明显的问题是,非字符串数据,比如一个邮编,明智的做法是明确地指定数据存储的角色为 `Qt::EditRole`,这样在 `Qt::DisplayRole` 被请求的时候令 Qt 自动产生一个相应的字符串来呈现数据(在表 3.2 中列出了所有的角色)。

一旦新添加了数据,就应滚动表格视图到底部(因为新邮编添加到了最末尾),并在新邮编的第一列发起编辑,这样用户就能立即在一个数字微调框中定位到这条数据(实际上它是一个经过简单自定义的 `QSpinBox` 的派生类)。

这里并没有调用 `setDirty()` 方法,也没有连接 `QStandardItemModel::rowsInserted()` 信号给任何槽函数。这样做的意思是,在新加载或新保存的数据时,应用程序并不认为自己有任何未保存的改变,然而,如果用户编辑任何一个新加的邮编单元格(或者其他任何单元格)就会发射信号 `itemChanged()`,而信号 `itemChanged()` 已经连接到 `setDirty()` 槽函数上。

`deleteZipCode()` 方法比较长,这里分成两部分来讲解。

```

void MainWindow::deleteZipcode()
{
    QItemSelectionModel *selectionModel = tableView->selectionModel();
    if (!selectionModel->hasSelection())
        return;
    QModelIndex index = proxyModel->mapToSource(
        selectionModel->currentIndex());
    if (!index.isValid())
        return;
    int zipcode = model->data(model->index(index.row(),
                                           Zipcode)).toInt();
    if (!AQP::okToDelete(this, tr("Delete Zipcode"),
        tr("Delete Zipcode %1?").arg(zipcode, 5, 10, QChar('0'))))
        return;
}

```

如果用户要删除行,首先要看用户是否选中了某些单元(cell),是否能把选集模型中的索引通过 `QSortFilterProxyModel::mapToSource()` 方法转化为底层模型中的正确索引。

一旦知道了用户想要删除的行(也就是所选单元格在底层模型索引所在的行),就需要提示用户进行删除确认,如果用户改变了主意就直接返回。

```
bool filtered = filterRadioButton->isChecked();
bool selected = selectByCriteriaRadioButton->isChecked();
QString county = countyGroupBox->isChecked()
    ? countyComboBox->currentText() : QString();
QString state = stateGroupBox->isChecked()
    ? stateComboBox->currentText() : QString();
dontFilterOrSelectRadioButton->click();

model->removeRow(index.row(), index.parent());

createComboBoxModels();
if (!county.isEmpty())
    countyComboBox->setCurrentIndex(
        countyComboBox->findText(county));
if (!state.isEmpty())
    stateComboBox->setCurrentIndex(
        stateComboBox->findText(state));
if (filtered)
    filterRadioButton->click();
else if (selected)
    selectByCriteriaRadioButton->click();
}
```

在删除行之前,先要保存当前界面中的各种过滤和选择状态,然后通过点击“Don't Filter or Select”单选按钮的方法来关闭过滤和选择状态。在删除行操作完成后,要恢复先前的过滤和选择状态。严格地说,并不需要保存和恢复过滤/选择状态,但我们需要重新创建组合框的模型,因为它们可能含有已经被删除的行的内容。例如,如果用户删除的行包含有数据集中唯一的某个县或州。

对于一个应用程序来说通常允许用户进行删除操作,为方便起见,我们提供了一个自定义的 `AQP::okToDelete()` 函数,下面把它完整地写出来。

```
bool okToDelete(QWidget *parent, const QString &title,
    const QString &text, const QString &detailedText)
{
    QScopedPointer<QMessageBox> messageBox(new QMessageBox(parent));
    if (parent)
        messageBox->setWindowModality(Qt::WindowModal);
    messageBox->setIcon(QMessageBox::Question);
    messageBox->setWindowTitle(QString("%1 - %2")
        .arg(QApplication::applicationName()).arg(title));
    messageBox->setText(text);
    if (!detailedText.isEmpty())
        messageBox->setInformativeText(detailedText);
    QAbstractButton *deleteButton = messageBox->addButton(
        QObject::tr("&Delete"), QMessageBox::AcceptRole);
    messageBox->addButton(QObject::tr("Do &Not Delete"),
        QMessageBox::RejectRole);
    messageBox->setDefaultButton(
        qobject_cast<QPushButton*>(deleteButton));
    messageBox->exec();
    return messageBox->clickedButton() == deleteButton;
}
```

在这个方法中除了返回值之外,其他的都和 `AQP::information()` 和 `AQP::warning()` 函数相同,并且创建和设置信息框(message box)与它们很相似。参数 `detailedText` 的默认值是空字符串(`QString()`),因此它可以被调用者省去(请参阅题为“避免 Qt 的静态便利 `QMessageBox` 函数”阴影部分的内容,即为什么使用自定义信息框函数)。`qobject_cast<>()` 调用是必要的,因为 `QMes-`

`sageBox::setDefaultButton()`方法需要一个 `QPushButton` 类型的指针作为参数,但我们声明的 `deleteButton` 是 `QAbstractButton` 类型的(为了与 `QMessageBox::clickedButton()` 函数返回值的比较更加方便,因为这个方法返回了一个 `QAbstractButton` 类型的指针)(在题为“Qt 的智能指针”阴影部分中将介绍 Qt 4.6 中的 `QScopedPointer` 类)^①。

理论上可以通过调用 `QWidget::setAttribute(Qt::WA_DeleteOnClose)` 方法来使得对话框在关闭后自动销毁,但在实践中建议把这个工作留给智能指针去处理。这就意味着信息框在点击按钮时,仅仅是被关闭了而没有被销毁,因此在 `QMessageBox::exec()` 方法调用返回后,信息框仍然存在。正如上面代码中所做的,如果我们想检查哪个按钮被按下了,保留对话框不被销毁的状态尤其重要。

要记住至少对于大型工程而言,使用 `QDialog::exec()` 方法已经过时了。问题在于,尽管从用户的角度来看它是一个阻塞式调用(也就是说,它阻止了用户与应用程序中其他窗体的交互),但它并没有阻塞事件处理。这就意味着程序的状态可以在调用 `exec()` 方法和用户接受或拒绝对话框的动作之间发生重大改变,甚至连对话框本身意外地被删除也是有可能的。考虑到这个问题,使用 `QDialog::open()`(或者对于非模态对话框使用 `QDialog::show()`),并且使用信号-槽连接来处理用户对于对话框的响应是更安全的方法。尽管如此,当我们接受使用 `exec()` 方法的问题和潜在的危险时,本书中的例子并无这个烦恼,因此我们会继续使用它,尤其是它比使用 `open()` 和使用信号连接的方法更加便利而且代码量更少^②。

现在我们了解了在用户界面中提供用户可操作模型的交互行为的所有相关方法。在下一小节将介绍用来加载、编辑、保存应用程序数据的 `QStandardItemModel` 派生类。在之后的两个小节中,将介绍用于过滤数据以确保用做过滤和选择的组合框总是保持唯一值的 `QSortFilterProxyModel` 派生类。

3.2.2 用于表格的 `QStandardItemModel` 子类

`QStandardItemModel` 类提供了操作表格数据及与视图交互的所有功能。为了处理真实的数据——即允许用户创建新的数据集,仅需补充的一点是需要加入从文件中载入数据和保存数据到文件中以及清除数据的功能。

下面从构造函数和 `clear()` 函数开始介绍,后面再来了解如何载入和保存数据。

```
StandardItemModel::StandardItemModel(QObject *parent)
    : QStandardItemModel(parent)
{
    initialize();
}

void StandardTableModel::initialize()
{
    setHorizontalHeaderLabels(QStringList() << tr("Zipcode")
        << tr("Post Office") << tr("County") << tr("State"));
}
```

因为在 `clear()` 方法中要用到 `initialize()` 函数的功能,所以这里把它单独分离出来。

```
void StandardTableModel::clear()
{
    QStandardItemModel::clear();
    initialize();
}
```

① 在源代码中我们使用了 `#if QT_VERSION` 以便使得用到的 `QSharedPointer` 的代码在 Qt 4.5 中能够编译。

② 请参考《Qt 季刊》第 30 期的“New Ways of Using Dialogs”一文,请访问 qt.nokia.com/doc/qq/ 以获得更多信息。

这里我们注意到这个方法(实际上也就是 `StandardTableModel` 类本身),它既不知道也不关心模型中的数据是否修改过,因此处理未保存的修改的事情就留给了类的使用者主窗体类来完成。

在这个派生类中只留下 `load()` 和 `save()` 两个方法没有介绍了。用于保存邮编数据的文件(其格式如图 3.4 所示)是一个磁盘上的二进制文件。文件内容以一个幻数(magic number)和一个表示文件格式的版本号开始,然后紧接着的是若干个邮编记录,每一条记录都有邮编、邮局、县和州信息。

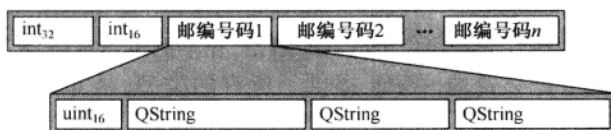


图 3.4 Zipcode 应用程序的文件的格式

```
const quint32 MagicNumber = 0x5A697043;
const quint16 FormatNumber = 100;

void StandardTableModel::save(const QString &filename)
{
    if (!filename.isEmpty())
        m_filename = filename;
    if (m_filename.isEmpty())
        throw AQP::Error(tr("no filename specified"));
    QFile file(m_filename);
    if (!file.open(QIODevice::WriteOnly))
        throw AQP::Error(file.errorString());

    QDataStream out(&file);
    out << MagicNumber << FormatNumber;
    out.setVersion(QDataStream::Qt_4_5);

    for (int row = 0; row < rowCount(); ++row) {
        out << static_cast<quint16>{
            item(row, Zipcode)->data(Qt::EditRole).toUInt()
        } << item(row, PostOffice)->text()
        << item(row, County)->text() << item(row, State)->text();
    }
}
```

保存数据非常简单。函数参数 `filename` 默认是空字符串,如果 `filename` 不为空,就使用参数 `filename` 所代表的文件,否则仍然使用上次使用过的那个文件。如果要打开的文件仍然为空字符串,或者打开失败,就抛出异常。如果抛出了异常,那么调用该方法的 `MainWindow::save()` 方法会捕获这个异常,并弹出一个信息框来显示这条错误信息。

文件一打开,首先就将幻数和文件格式标志写进去。这里使用一个唯一的(希望如此)的幻数来标志邮编文件格式,并使用一个数字来标志我们使用的文件版本号(在“MIME 类型文件后缀和幻数”的内容中有关幻数的简要介绍)。使用文件格式标志,使得在之后如想变更文件格式的时候,会变得容易些,这是因为我们可以根据版本号载入文件并按版本做相应调整。在写完这些信息后,设置 `QDataStream` 的版本为 `Qt_4_5`(`Qt 4.5` 及更高版本可读取),最后写入邮编记录。由于数据是由数字和字符串组成的,我们可以很容易地在较老的 `Qt` 版本(比如 `Qt_4_0`)中使用这个文件。使用最新的 `QDataStream` 版本号的唯一优点是新版本可以比老版本支持更多的 `Qt` 数据类型,或者在读、写时更快,或者存储的数据更紧凑。我们的原则是使用能够兼容编译这个应用程序示例的最低的 `Qt` 版本。

基础数据类型如整型 (integer) 等不论 QDataStream 对象所使用的 Qt 版本是什么, 都拥有相同的数据格式, 所以不设置版本号, 即读取这些类型也总是安全的。然而, Qt 特有的类型以及浮点数据类型的数据格式可能在不同的 QDataStream 版本中是不同的, 因此, 我们必须总是确保所读写的 Qt 特有类型以及浮点数据类型应使用相同的 QDataStream 版本^①。这里要注明的是, 永远不要使用 qreal 类型来进行浮点数据类型的读写, 因为 qreal 的长度是平台相关的 (必须明确地使用 float、double 类型)。

对于字符串数据, 可以直接从每一个 QStandardItem 对象的 text 属性中获取, 但是对于非字符串数据 (例如邮编), 必须使用存储在角色中的真实数据, 在本例中通常是 Qt::EditRole。写整型数据时使用正确的数据类型是必要的——Qt 提供了所有的类型, 从 qint8 和 quint8 到 qint64 和 quint64。

保存和加载 QStandardItem 对象数据的另一个方法是使用流操作, 因为它支持将 QDataStream 对象作为 Qt 的全局操作符 operator <<() 和 operator >>() 的第一个参数。可以通过创建 QStandardItem 的派生类并重新实现流操作符要调用的 QStandardItem::read() 和 QStandardItem::write() 方法来获得更细致的读、写控制, 然而在本例中, 我们选择使用现成的 QStandardItem, 并在 QStandardItemModel 派生类中处理对它们的读、写。

这里用于抛出异常的 Error 类来源于 aqp. {hpp, cpp} 模块, 下面给出它的完整内容。

```
class Error : public std::exception
{
public:
    explicit Error(const QString &message) throw()
        : message(message.toUtf8()) {}
    ~Error() throw() {}

    const char *what() const throw() { return message; }

private:
    const char *message;
};
```

Error 类没有什么特别的地方, 只是使用了更方便的 QString 类型而不是 char * 类型来传递错误信息。相应地, 必须使用 QString::fromUtf8(error.what()) 来获取错误信息。

load() 函数方法要比 save() 方法长一些, 所以这里将分成 4 小段进行介绍。

```
void StandardTableModel::load(const QString &filename)
{
    ...
    QDataStream in(&file);
    qint32 magicNumber;
    in >> magicNumber;
    if (magicNumber != MagicNumber)
        throw AQP::Error(tr("unrecognized file type"));
    qint16 formatVersionNumber;
    in >> formatVersionNumber;
    if (formatVersionNumber > FormatNumber)
        throw AQP::Error(tr("file format version is too new"));
    in.setVersion(QDataStream::Qt_4_5);
    clear();
}
```

load() 方法的开头部分的代码与 save() 方法中关于使用已有的文件名或新文件名部分的代码是一模一样的, 所以这里省略掉了这部分代码。当打开参数 filename 所指的文件后 (这次使用了

^① 在 Qt 4.5 和 Qt 4.6 中 float 数据类型的表达方式有所变化, 倘若这样使用 QDataStream::setVersion() 方法, 像这样的变化是没有问题的, 并且对代码或数据也没有什么影响。

QIODevice::ReadOnly 打开模式)。先读取幻数来检查读取的是不是一个邮编数据文件,然后读取文件格式版本数字。在本例中,仅检查格式版本,但是如果有需要,应该在这里正确地处理不同版本的文件读取问题。之后设置正确的 QDataStream 版本,并清除模型以确保模型中的数据都已被删除。

```
quint16 zipcode;
QString postOffice;
QString county;
QString state;
 QMap<quint16, QList<QStandardItem*>> > itemsForZipcode;
```

通常在读取记录的时候就可以把每一条数据插入到模型中。但因为我们想先对数据进行排序,所以这里使用了一个按邮编排序的 QMap 对象来临时保存每一条记录(保存 QStandardItem 指针的一个 QList 对象),在最后把这些记录加入到模型中。

```
while (!in.atEnd()) {
    in >> zipcode >> postOffice >> county >> state;
    QList<QStandardItem*> items;
    QStandardItem *item = new QStandardItem;
    item->setData(zipcode, Qt::EditRole);
    items << item;
    foreach (const QString &text, QStringList() << postOffice
                                                    << county << state)
        items << new QStandardItem(text);
    itemsForZipcode[zipcode] = items;
}
```

在读取每一条记录时,创建一个对应的 QList<QStandardItem *> 对象。创建 QStandardItem 对象时,字符串数据只需简单地传给构造函数即可,但对于非字符串数据,如邮编,则应设置对应的角色数据,通常是 Qt::EditRole(如本例的邮编数据)。当已经获取到了代表一行记录的所有 QStandardItem 对象时,就可以把它插入到 QMap 对象中,使用邮编作为索引键,以此来实现按邮编(从小到大)的排序。

```
QMapIterator<quint16, QList<QStandardItem*>> > i(itemsForZipcode);
while (i.hasNext())
    appendRow(i.next().value());
}
```

当所有的记录都读取到后,通过遍历 QMap 对象,把每一个 QList<QStandardItem *> 对象作为新行加入到模型中。在这里不必通知相关联的视图关于模型数据的变化,因为基类的 appendRow() 方法已经进行了处理。

在先前提到过,在我们的测试机上加载数据到 QStandardItemModel 对象总是比加载数据到自定义模型中要慢。当直接加载数据时(即不用 QMap 对象来进行排序处理)也是如此,因此在机器上所显现出来的耗时操作,实际上是在于创建每个 QStandardItem 的这一步之上。

现在已经介绍完了 StandardItemModel 派生类。因为在 QStandardItemModel 中唯一增加的功能就是加载、保存和清除数据,并且由于处理错误信息和处理未保存的数据变更操作的工作留给了使用它的类(即主窗体类),所以这个 QStandardItemModel 派生类实现起来相当简单。在下一节将创建一个 QAbstractTableModel 派生类做一简单替换,以便能对这两种实现方法进行比较。但首先应看一下那些在应用程序的 QTableView 部件和 QComboBox 部件中用到的 QSortFilterProxyModel 派生类。

3.2.3 过滤掉重复行的 QSortFilterProxyModel 类

Zipcodes 应用程序使用了两个组合框来分别显示县和州,这样用户就能从全部数据中进行过滤或选择操作。这两个组合框都需要从含有所有邮编数据的底层模型获得自己的数据——但许

多邮编有可能属于同一个县和州,我们不想在这两个组合框中显示重复的数据,因此不能设置组合框直接去使用底层模型。

一个解决方法是创建一个能将模型中所有某列重复的行都过滤出去的自定义 `QSortFilterProxyModel` 派生类。要进行过滤操作,需要实现保护方法 `filterAcceptsRow()` 来过滤行,或者实现 `filterAcceptsColumn()` 方法来过滤列(对于排序,需要实现保护方法 `QSortFilterProxyModel::lessThan()`,尽管简单地使用 `QSortFilterProxyModel` 类提供的排序功能即可——这是我们在 `Zipcodes` 程序中所倚仗的功能,将在后面进行说明)。在这个特定示例中我们还需要重新实现即将要介绍的 `setSourceModel()` 方法。首先来看一下头文件中的定义。

```
class UniqueProxyModel : public QSortFilterProxyModel
{
    Q_OBJECT
public:
    explicit UniqueProxyModel(int column, QObject *parent=0)
        : QSortFilterProxyModel(parent), Column(column) {}

    void setSourceModel(QAbstractItemModel *sourceModel);

protected:
    bool filterAcceptsRow(int sourceRow,
                          const QModelIndex &sourceParent) const;

private slots:
    void clearCache() { cache.clear(); }

private:
    const int Column;
    mutable QSet<QString> cache;
};
```

这里声明了一个 `mutable` 变量(`mutable` 这个关键字表示其所修饰的值是经常变化的) `cache` 保存在特定列中 `UniqueProxyModel` 能看到的所有的唯一字符串。

```
void UniqueProxyModel::setSourceModel(
    QAbstractItemModel *sourceModel)
{
    connect(sourceModel, SIGNAL(modelReset()),
            this, SLOT(clearCache()));
    QSortFilterProxyModel::setSourceModel(sourceModel);
}
```

如果源模型(即底层模型 `model`)重置(`reset`)了,就要清除 `cache`,因为数据已经从根本上改变了。应该添加将插入或删除行的信号与清除 `cache` 的槽函数 `clearCache()` 连接起来的代码吗?对于插入行,没有必要清除 `cache`,因为通过实现 `filterAcceptsRow()` 方法已经把这些字符串正确地处理了。对于删除行,是否应删除 `cache` 中对应的数据,取决于我们是不是想让用户访问到那些本已不存在的数据,但在本例中是允许用户访问的。

```
bool UniqueProxyModel::filterAcceptsRow(int sourceRow,
    const QModelIndex &sourceParent) const
{
    QModelIndex index = sourceModel()->index(sourceRow, Column,
                                              sourceParent);
    const QString &text = sourceModel()->data(index).toString();
    if (cache.contains(text))
        return false;
    cache << text;
    return true;
}
```

每一个组合框都设置自己的模型为一个 `UniqueProxyModel` 实例,将列作为对应的行(县州);这在 `cre-`

ateComboBoxModel()方法中处理。无论何时当组合框需要访问这些数据行的时候(例如当用户点击右侧的下拉按钮弹出组合框的列表视图时),这个方法就可以用来把代理模型中不需要的行过滤掉。

算法非常简单:先获取代理模型的行对应的底层模型(model)中的索引,然后检索底层模型中对应列(在代理模型创建时设定)的内容,如果这个内容在 cache 中已经存在(以前添加的)就返回 false,表示这一行被过滤掉了。否则就把这个内容添加到 cache 中(后面的行如果指定列有相同的内容,就会被过滤掉),然后返回 true 以允许该行显示出来且仅此一次。

3.2.4 在已有数据中进行过滤的 QSortFilterProxyModel

上一小节介绍的 UniqueProxyModel 非常有用,但只是针对一个特定用例的。对于 Zipcodes 应用程序来说,我们需要一个更加高级的、允许用户在组合条件——可按邮编的最大值、最小值限定,可以选择按县、州进行过滤之后再行过滤的代理模型。我们新建一个具有这些功能的类 ProxyModel,如 UniqueProxyModel 一样,重新实现 filterAcceptsRow()方法完成过滤。

ProxyModel 派生类在构造函数之外,还提供了各种过滤条件的获取(getter)和设定(setter)方法、filterAcceptsRow()方法,以及 clearFilters()方法。除了那些获取函数和设定函数之外的方法这里都将进行介绍,对于获取和设定函数,因为它们在结构上都是相同的,所以这里只挑其中之一进行介绍。

```
ProxyModel::ProxyModel(QObject *parent)
    : QSortFilterProxyModel(parent)
{
    m_minimumZipcode = m_maximumZipcode = InvalidZipcode;
}
```

首先初始化代表邮编的最小值和最大值的成员变量(m_minimumZipcode、m_maximumZipcode)为无效邮编的常量 InvalidZipcode,在 filterAcceptsRow()方法中用于确定是否可以略过邮编的对比(即不过滤)。

```
QString state() const { return m_state; }
```

这个获取函数在头文件中进行了定义;县(county)和最大、最小邮编值的获取方法也与此相似。

```
void ProxyModel::setState(const QString &state)
{
    if (m_state != state) {
        m_state = state;
        invalidateFilter();
    }
}
```

所有的设定函数都与上面的代码有相同的模式:如果给定值确实发生了变化,首先设置值,然后调用 invalidateFilter(),以使代理模型向所有需要刷新可见数据的关联视图发布通知。

如果给定的是一个空字符串,等效于关闭按州过滤条件。对于县的设定函数来说也是同样的道理。

```
void ProxyModel::clearFilters()
{
    m_minimumZipcode = m_maximumZipcode = InvalidZipcode;
    m_county.clear();
    m_state.clear();
    invalidateFilter();
}
```

现在,如果邮编最大值和最小值是无效的,或县和州字符串是空的,就没有任何行被过滤掉了,因此调用这个方法等效于关闭了过滤功能。

```

bool ProxyModel::filterAcceptsRow(int sourceRow,
    const QModelIndex &sourceParent) const
{
    if (m_minimumZipcode != InvalidZipcode ||
        m_maximumZipcode != InvalidZipcode) {
        QModelIndex index = sourceModel()->index(sourceRow, Zipcode,
                                                    sourceParent);
        if (m_minimumZipcode != InvalidZipcode &&
            sourceModel()->data(index).toInt() < m_minimumZipcode)
            return false;
        if (m_maximumZipcode != InvalidZipcode &&
            sourceModel()->data(index).toInt() > m_maximumZipcode)
            return false;
    }
    if (!m_county.isEmpty()) {
        QModelIndex index = sourceModel()->index(sourceRow, County,
                                                    sourceParent);
        if (m_county != sourceModel()->data(index).toString())
            return false;
    }
    if (!m_state.isEmpty()) {
        QModelIndex index = sourceModel()->index(sourceRow, State,
                                                    sourceParent);
        if (m_state != sourceModel()->data(index).toString())
            return false;
    }
    return true;
}

```

当邮编最小值和最大值为无效并且县和州为空时,则该函数返回 true(保留行)。当邮编值大于等于邮编最小值的时候,它是有效的。

当邮编最小值有效并且当前行的邮编值比邮编最小值更小,则函数返回 false 以过滤掉该行。同样地,如果邮编最大值有效,并且当前行比邮编最大值更大,则函数返回 false 以过滤掉该行。如果代表县的值不为空,则该函数将所有属于其他县的行过滤掉,同样适用于州。如果到达了函数末尾,则直接返回 true(保留当前行)。

如该函数以及前一小节中的同名方法所示,创建一个具有过滤功能的自定义 QSortFilterProxy-Model 派生类是不难的。当然可以将这些过滤条件串联起来——只是在性能上付出些代价。

子类化并不是使用 QSortFilterProxyModel 的唯一方法。仍然能够直接实例化 QSortFilterProxy-Model 类,并使用 setFilterKeyColumn() 方法来选择过滤的列,使用 setFilterRegExp() 方法来设置一个正则表达式来进行过滤,指定列的内容与正则表达式不匹配的行,将被过滤出去(也可以使用普通字符串和通配符模式)。

对于排序可以采用多种方法。当视图接收到排序请求时,幕后的 QAbstractItemModel::sort() 方法将被调用,因此一种排序方法是重新实现 QAbstractItemModel::sort() (因为基类的实现没有做任何事情)。对于 Zipcodes 应用程序来说,视图调用 QSortFilterProxyModel::sort() 方法(因为视图的模型是一个代理模型)进行排序,它有一个默认的能进行整型及一些基本的 Qt 类型的排序,比如 QString 和 QDateTime。这就是为什么对 Zipcodes 应用程序的模型提供排序功能,只需要使用 QSortFilterProxyModel 类,并将视图的横表头(horizontal header)的 sectionClicked() 信号与视图的 sortByColumn() 槽连接起来就可以了的原因。

我们还可以通过调用 setSortCaseSensitivity() 和 setSortLocaleAware() 方法对代理模型的排序实施更精细的控制;或者还可以通过子类化 QSortFilterProxyModel 并重新实现 lessThan() 的方法来处理排序。

对于 `QStandardItemModel` 派生类来说,另一个排序方法是使用 `QStandardItemModel::setSortRole()` 方法,例如,设置 `Qt::UserRole` 对应的数据为排序用的数据。要想这种方式能够正常工作,必须调用 `setSortingEnabled(true)` 方法来告知视图以支持排序,并且必须确保对于每一项(item),除了供显示、编辑角色使用的数据之外,还要有用户角色(user role)的数据(或者我们设置为排序角色的任意角色)用来排序。假设,例如,有若干显示为英文的项,可能在 `Qt::DisplayRole` 角色中保存真实的内容,但对于 `Qt::UserRole` 角色可能保存的是相同的内容[但却是小写的,并且已去除任何前导冠词(The、An 和 A)],以提供更加自然的排序功能。

QStandardItemModel 和自定义模型的比较

当我们使用 `QStandardItemModel` 或一个 `QStandardItemModel` 的子类来代表数据时(无论是一个列表、表格还是树),所有的数据项都保存为 `QStandardItem` 项(或自定义的 `QStandardItem` 子类的项)。

从概念上说,`QStandardItemModel` 对象的定位介于视图部件加上模型的方式与便利窗体和内置模型部件的方式之间。使用 `QStandardItemModel` 要比创建自定义模型容易一些,又要比便利窗体部件更灵活,因为我们经常能直接使用 `QStandardItemModel`,甚至在需要子类化(`QStandardItemModel` 类)的时候,通常仅仅添加少量方法(例如加载、保存)就可以了。另一个某些开发者比较喜欢 `QStandardItemModel` 的因素是,它使用一个更加熟悉的基于项的 API 而不是自定义模型的基于索引的 API。

`QStandardItem` 类提供了丰富的 API 接口,可以使得 `QStandardItem` 对象非常方便地去直接使用。一个 `QStandardItem` 对象的最常用函数是提供项的背景色(background color)、是否能选择(checkability)、选中状态(checked status)、是否可编辑(editability)、字体(font)、前景色(foreground color)、图标(icon)、状态提示(status tip)、文本(text)、文本对齐方式(text alignment)和工具提示(tooltip)的属性的获取和设定方法。额外的数据可以保存在 `QStandardItem` 对象的未使用的角色中,例如 `Qt::UserRole`、`Qt::UserRole + 1` 诸如此类。我们甚至可以把 `QStandardItem` 对象从 `QDataStream` 流中读入或者输出到 `QDataStream` 流中。

`QStandardItem` 对象为所有的便利性和强大能力所付出的代价(至少是理论上的)是内存消耗,或许还有操作速度。一个自定义模型或许根本不需要存储每个单独的项,或者是仅需一些像字符串或数字的轻量级的项。

在所有的情况下,使用 `QStandardItemModel` 和 `QStandardItem`(或它们的派生类)通常是最容易和最快捷的方法,至少在一开始时是这样。使用这些类允许我们快速地完成一个可工作的原型程序。如果过些时候发现内存消耗或操作速度不令人满意,我们就可以考虑创建一个自定义模型来代替它。

哪些数据项是轻量级的、不需要那么多的 `QStandardItem` 类提供的特性的,以及数据项特别多的(数以千计或更多)应用程序是最可能从自定义模型中获得好处的。并且,对于树模型, `QStandardItemModel` 的 API 无法提供我们用自定义模型可以实现的那么多功能。

创建自定义列表和表格模型比较简单,因此它们是在大数据集的情况下潜在地获取一个性能优势的方法。自定义树模型需要做相当多的工作,并且要处理正确是相当复杂的,但是它对于仅仅是为了获取额外的、只有用自定义模型才能提供的功能而言可能是必要的。例如,经常性地树中移动项(包括其子项,递归),当然,这对于我们创建的特别数据集而言是有意义的。

3.3 创建自定义表格模型

就 Qt 中表现的数据而言,使用 `QStandardItemModel` 通常是最简单和最便捷的方式。然而,对于特定数据来说,`QStandardItemModel` 类使用的 `QStandardItem` 对象可能没有必要那么笨重(例如,消耗了更多的内存),因为它们更多的是为通用目的而非我们的特殊用例而设计的。

在本节中,我们将在创建 `zipcodes2` 应用程序的过程中,使用一个自定义模型来代替 `zipcodes1` 应用程序所使用的简单 `QStandardItemModel` 派生类。这两个应用程序都有相同的外观和行为表现,但 `zipcodes2` 在我们的测试机上加载数据时比较快些。

这个新应用程序新增三个文件:`zipcodeitem.h`、`tablemodel.h` 和 `tablemodel.cpp`。除了不再需要的 `standardtablemodel.h`、`tablemodel.h`、`tablemodel.cpp` 外,其他文件都与 `zipcodes1` 里的相同。工程文件 `zipcodes2.pro` 增加了一行内容“`DEFINES += CUSTOM_MODEL`”,同时 `zipcodes1` 应用程序里的文件中,在与 `zipcodes2` 应用程序的实现有必要进行分别处理的地方,都使用 `#ifdef` 编译预处理指令。

由于主窗体里的大多数方法在 `zipcodes1` 和 `zipcodes2` 应用程序里的是一样的,所以这里将只介绍两者有所区别的方法——构造函数、`createConnections()` 和 `addZipcode()`。这里将不显示 `#ifdef` 指令,而仅显示当 `CUSTOM_MODEL` 宏定义时,`zipcodes2` 应用程序中能够编译到的代码(在前一节中也是这样做的,只显示在没有定义 `CUSTOM_MODEL` 宏时编译器处理到的代码)。

3.3.1 通过用户界面改变表格模型

`zipcodes2` 应用程序的构造函数与 `zipcodes1` 的构造函数几乎相同,但值得再介绍一下,因为这里有我们前面没有讲到的内容。

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), loading(false)
{
    model = new TableModel(this);
#ifdef MODEL_TEST
    (void) new ModelTest(model, this);
#endif
    proxyModel = new ProxyModel(this);
    proxyModel->setSourceModel(model);

    createWidgets();
    createComboBoxModels();
    createLayout();
    createConnections();

    AQP::accelerateWidget(this);
    setWindowTitle(tr("%1 (Custom Model)[*]")
        .arg(QApplication::applicationName()));
    statusBar()->showMessage(tr("Ready"), StatusTimeout);
}
```

与 `zipcodes1` 不同的地方是这里使用的是 `TableModel` 类(`QAbstractTableModel` 的派生类),而不是一个 `QStandardItemModel` 派生类,并且初始化窗体标题内容也不同。

对于这两个应用程序而言,都创建了一个 `ModelTest` 对象(尽管在前面没有介绍)。这个对象用来对模型进行测试,可以从 Qt 开发框架的 `labs.qt.nokia.com/page/Projects/Itemview/Modeltest` 网址来获得。它非常容易使用,在下载完后,把它加入到当前项目的同级目录中,并把它加入到当前的工程 `.pro` 文件中。下面是 `zipcodes2.pro` 文件中的相关行(对于 `zipcodes1.pro` 也一样):

```
exists(..modeltest-0.2/modeltest.pri) {
    DEFINES += MODEL_TEST
    include(..modeltest-0.2/modeltest.pri)
}
```

.proFile

这里使用了 `qmake` 的 `exists()` 函数来判断指定文件是否存在, 如果存在则定义了 `MODEL_TEST` 宏, 然后使用 `include()` 函数将这个 `.pri` 文件包含进来。还必须在 `.pro` 文件中加入“`CONFIG += debug`”这行内容, 否则 C++ 编译预处理器将去除掉那些模型测试所用到的 `Q_ASSERT` 宏调用。

当我们想创建 `ModelTest` 对象时, 还要在源文件中将头文件包含进去, 在本例的 `mainwindow.cpp` 文件中:

```
#ifndef MODEL_TEST
#include <modeltest.h>
#endif
```

这里把模型测试相关的代码用 `#ifdef` 宏包含起来, 为的是应用程序在模型测试不存在的情况下也能进行编译。

最后一步创建一个 `ModelTest` 实例, 并把要测试的模型传递给它——在主窗体的构造函数中已经进行了处理。不需要再做任何事情, 当使用一个自定义模型且出错时(只要 `ModelTest` 能够识别得到), `ModelTest` 对象将会诊断并通知我们问题所在。对于就像将要创建的自定义模型来说, 工作相当简单, 看似并不真正地需要 `ModelTest`(但是对于像将要在下一章中创建的树模型来说, 真正值得使用 `ModelTest`)。

对比两个 `Zipcodes` 应用程序的 `createConnections()` 方法, 只有一个不同的地方, 这就是在 `zipcodes1` 中是将 `QStandardItemModel` 的 `itemChanged()` 信号与 `setDirty()` 连接了起来, 而在 `zipcodes2` 应用程序中则被替换为将自定义的 `TableModel` 的 `dataChanged()` 信号与 `setDirty()` 连接起来。

在两个应用程序中唯一差别较大的方法是 `addZipcode()`, 在 `zipcodes2` 应用程序中要稍微简单一些(在 `zipcodes1` 中对应的实现在前面介绍过)。

```
void MainWindow::addZipcode()
{
    dontFilterOrSelectRadioButton->click();
    if (!model->insertRow(model->rowCount()))
        return;
    tableView->scrollToBottom();
    tableView->setFocus();
    QModelIndex index = proxyModel->index(proxyModel->rowCount() - 1,
                                           Zipcode);
    tableView->setCurrentIndex(index);
    tableView->edit(index);
}
```

不像在 `zipcodes1` 中的版本, 在这里添加新行时不需要创建所有的项。而是仅仅调用了 `QAbstractTableModel::insertRow()` 方法, 这相应调用了以 1 为计数(作为参数)的 `insertRows()` 方法。对于允许插入行的自定义模型来说, 不管怎样必须实现 `insertRows()` 方法, 所以这里没有涉及到额外的工作。

两个应用程序都具有的所有其他的类和方法都是相同的, 所以自定义委托和两个自定义代理模型(`UniqueProxyModel` 和 `ProxyModel`)都没有变化, 两个应用程序的所有行为都是一样的, 在使用 `QStandardItemModel` 派生类和 `QAbstractTableModel` 派生类之间没有什么可区分的不同之处。当然在幕后, 创建一个 `QAbstractTableModel` 必须做更多的工作(因为 `QStandardItemModel` 提供了许多额外的功能可直接使用), 但是这样能获得更好的控制和潜在的更佳性能。

对视图或委托来说, 把一个模型替换成另一个(以处理相同的数据为前提), 不需要做任何改变, 实际上它突出了一个使用 Qt 的模型/视图架构的关键优点。

3.3.2 用于表格的 `QAbstractTableModel` 子类

在这一小节中将介绍如何创建一个 `QAbstractTableModel` 派生类。就像在 Qt 的模型/视图架构中所有的 `QAbstractItemModel` 派生类一样, 必须实现一组特定的函数, 以使得我们的模型派生类的 API 与架构兼容, 而且可以应用在任何需要这种模型的地方。

表 3.1 列出了在几种不同情况下必须要实现的方法。举个例子,所有的模型必须实现那些提供数据读取的方法(`flags()`、`data()`,等等),可编辑的模型必须实现那些提供数据读取支持的方法和那些支持编辑操作的方法,诸如此类(提供拖放支持必须要实现的方法,这里没有列出——我们将在下一章介绍这些内容)。

表 3.1 QAbstractItemModel 的 API

方 法	说 明
所有模型	
<code>data(index, role)</code>	返回所属模型的指定项(索引为 <code>index</code>)的角色(<code>role</code>)(<code>Qt::ItemDataRole</code> 类型)所对应的 <code>QVariant</code> 类型数据值
<code>flags(index)</code>	返回表示指定项(索引为 <code>index</code>)是否可用、是否有复选框、是否可编辑、是否可选择等状态的一个或多个 <code>Qt::ItemFlag</code> 类型值的按位或
<code>headerData(sect,orient,role)</code>	返回由参数 <code>sect</code> 所指行(当 <code>orient</code> 为 <code>Qt::Vertical</code> 时)或列(当 <code>orient</code> 为 <code>Qt::Horizontal</code> 时)处参数 <code>role</code> 所指角色的 <code>QVariant</code> 类型的表头值
<code>rowCount(index)</code>	返回以参数 <code>index</code> 为父项的那些项的行数
表格和树模型	
<code>columnCount(index)</code>	返回以参数 <code>index</code> 为父项的那些项的列数(通常对于整个模型来说是常量)
所有可编辑的模型	
<code>setData(index,value,role)</code>	设置指定项(索引为 <code>index</code>)的指定角色(<code>role</code>)对应的数据值为 <code>value</code> ,如果成功则返回 <code>true</code> 并且发出 <code>dataChanged()</code> 信号
<code>setHeaderData(sect,orient,value,role)</code>	设置由参数 <code>sect</code> 所指的行(当 <code>orient</code> 为 <code>Qt::Vertical</code> 时)或列(当 <code>orient</code> 为 <code>Qt::Horizontal</code> 时)处的指定角色(<code>role</code>)对应的表头值为 <code>value</code> ,如果设置成功,则返回 <code>true</code> 并且发出 <code>headerDataChanged()</code> 信号
所有可添加/删除行的模型	
<code>insertRows(row,count,index)</code>	在指定项(索引为 <code>index</code>)的子项中,在索引为 <code>row</code> 的行之前插入 <code>count</code> 行;重新实现时必须调用 <code>beginInsertRows()</code> 和 <code>endInsertRows()</code>
<code>removeRows(row,count,index)</code>	在指定项(索引为 <code>index</code>)的子项中,从索引为 <code>row</code> 的行开始删除 <code>count</code> 行,如果成功则返回 <code>true</code> ;重新实现该方法时必须调用 <code>beginRemoveRows()</code> 和 <code>endRemoveRows()</code>
可添加、删除列的表格和树模型	
<code>insertColumns(column,count,index)</code>	在指定项(索引为 <code>index</code>)的子项中,在索引为 <code>column</code> 的列之前插入 <code>count</code> 列;如果成功则返回 <code>true</code> ;重新实现该方法时必须调用 <code>beginInsertColumns()</code> 和 <code>endInsertColumns()</code>
<code>removeColumns(column,count,index)</code>	在指定项(索引为 <code>index</code>)的子项中,从索引为 <code>column</code> 的列开始删除 <code>count</code> 列;如果成功则返回 <code>true</code> ;重新实现该方法时必须调用 <code>beginRemoveColumns()</code> 和 <code>endRemoveColumns()</code>
树模型	
<code>index(row,column,index)</code>	返回指定项(索引为 <code>index</code>)的子项中,行号为 <code>row</code> 、列号为 <code>column</code> 的项的索引(<code>QModelIndex</code> 类型)
<code>parent(index)</code>	返回参数 <code>index</code> 所指项的父项索引(<code>QModelIndex</code> 类型)

对于一些可进行插入/删除行或列的模型来说,需要提供几个有意义的方法进行操作。举个例子,一个表格模型可能允许对行而不允许对列进行插入和删除操作。在这种情况下就必须实现 `insertRows()` 和 `removeRows()` 方法,而不需要实现 `insertColumns()` 或 `removeColumns()` 方法。

在这一小节介绍的 `TableModel` 类是一个 `QAbstractTableModel` 派生类,它把数据存储在一个 `QList<ZipcodeItem>` 类型的成员对象中。选择 `QList` 而不是 `QVector`,是因为在通常情况下它能提供比 `QVector` 更好的性能,而对在中间进行插入和删除操作更是如此(对于真正的大列表,如果需要在中间进行插入、删除操作,`QLinkedList` 是个更好的选择)。

在 QList 中存储的数据项必须是一种可赋值的数据类型,换言之,即一个提供了默认构造函数、拷贝构造函数和赋值操作符的类型。此外,为了使用某些方法,在该类中提供一些额外的操作符是必要的。例如 QList::contains()、QList::count() (为特定值)、QList::indexOf()、QList::lastIndexOf()、QList::removeAll()、QList::removeOne() 或者 QList::operator!=(), 该数据项类型必须提供 operator==()。为了使用 qSort() 函数以支持排序功能,该数据项类型必须提供 operator<()。下面是在头文件中完整定义的 ZipcodeItem 类:

```
struct ZipcodeItem
{
    explicit ZipcodeItem(int zipcode=InvalidZipcode,
        const QString &postOffice=QString(),
        const QString &county=QString(),
        const QString &state=QString())
        : zipcode(zipcode_), postOffice(postOffice_), county(county_),
          state(state_) {}

    bool operator<(const ZipcodeItem &other) const
    { return zipcode != other.zipcode ? zipcode < other.zipcode
      : postOffice < other.postOffice; }

    int zipcode;
    QString postOffice;
    QString county;
    QString state;
};
```

这个类有一个默认构造函数——所有的参数都有默认值,所以调用时都可以省略。因为数据成员都是值,而非指针,我们让 C++ 本身来提供拷贝构造函数和赋值操作符。因为没有实现 operator==(), 所以无法在 QList 中进行搜索 ZipcodeItem 数据项或做任何依赖于这个操作符的操作。但是实现了 operator<()——进行邮编的比较并且在相同邮编值的情况下使用邮局字段进行二次比较,因为我们希望能够按邮编进行排序。

一个更加详尽的类应该使用到了获取函数和设定函数,或许还提供一个 isValid() 方法和 operator==(), 但是对于 TableModel 来说它们都不是必需的,所以这里没有实现它们。

类 TableModel 实现了所有支持数据读取的方法 (flags()、data()、headerData()、rowCount()、columnCount()), 还实现了提供编辑功能的两个方法 (setData() 和 setHeaderData()), 但只实现了两个进行插入/删除行的方法 (insertRows() 和 removeRows()), 因此 TableModel 的列数是固定的。此外,还提供了 load() 和 save() 方法,以及 filename() 获取方法,以提供从文件中加载数据和写数据到文件的功能(参见表 3.2)。

TableModel 类有两个私有数据成员: QList< ZipcodeItem > 类型的 zipcodes 和 QString 类型的 m_filename。构造函数仅仅调用基类 QAbstractTableModel 的构造函数,并把 parent 参数传递过去,构造函数体则是空的。

下面就让我们看一下 TableModel 类的方法。尽管它们需要访问的数据是专门针对于 Zipcodes 应用程序的,但它们的实现结构可以推广到任何 QAbstractTableModel 或 QAbstractItemModel 派生类中,因此尽管代码不能直接剪切、粘贴就能用,但它可以作为一个实现此类自定义表格模型的模型/视图方法的通用模板。

3.3.2.1 与表格相关的 QAbstractItemModel API 方法

我们先从介绍 QAbstractItemModel 的方法开始,它们是提供可编辑和扩展(可以对行,但不能对列)项的自定义表格模型所必须实现的方法。

表 3.2 Qt::ItemDataRole 枚举类型

枚 举 值	说 明
Qt::AccessibleDescriptionRole	关于辅助特性支持的描述内容
Qt::AccessibleTextRole	辅助工具(比如屏幕阅读器)所使用的文本内容
Qt::BackgroundRole	渲染数据时所使用的背景画刷
Qt::CheckStateRole	数据项的复选状态
Qt::DecorationRole	数据的代表图标
Qt::DisplayRole	将要表达的数据显示成文本
Qt::EditRole	宜于编辑格式的数据
Qt::FontRole	渲染数据为文本时所使用的字体
Qt::ForegroundRole	渲染数据时使用的前景画刷
Qt::SizeHintRole	数据的尺寸大小提示
Qt::StatusTipRole	数据的状态栏提示内容
Qt::TextAlignmentRole	渲染数据为文本时文本使用的对齐方式
Qt::ToolTipRole	数据的工具提示
Qt::UserRole	可存储自定义数据的角色;更多的数据可存储在 Qt::UserRole + 1 等角色中
Qt::WhatsThisRole	数据的“这是什么?”文本内容

```
Qt::ItemFlags TableModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags theFlags = QAbstractTableModel::flags(index);
    if (index.isValid())
        theFlags |= Qt::ItemIsSelectable|Qt::ItemIsEditable|
                    Qt::ItemIsEnabled;
    return theFlags;
}
```

如果提供给我们的模型索引是有效的,则设置相应项的 flags 以使该项能够被选择和编辑——当然也是可用(enabled)的。事实上,基类的实现返回了 Qt::ItemIsSelectable|Qt::ItemIsEnabled,因此这里只需要加上 Qt::ItemIsEditable 就可以了;但我们更倾向于明确地表达出我们的意图(在表 3.3 中列出了这些枚举值)。

表 3.3 Qt::ItemFlag 枚举类型

枚 举 值	说 明
Qt::ItemIsDragEnabled	可拖动标志
Qt::ItemIsDropEnabled	可放置标志
Qt::ItemIsEditable	可编辑标志
Qt::ItemIsEnabled	可与用户交互
Qt::ItemIsSelectable	可选择标志
Qt::ItemIsTristate	具有三种(而非两种)复选状态标志(选中、未选中、未改变)
Qt::ItemIsUserCheckable	具有用户可操作的复选框标志
Qt::NoItemFlags	无任何标志。如果这是唯一的标志,则该项不可复选、不可选择、不可编辑等

data()方法是所有关注数据和元数据的项(item)被访问的途径,尽管这个方法不长,我们仍然要将其分成 4 小段以便于进行讲解。

```
const int MaxColumns = 4;

QVariant TableModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid() ||
        index.row() < 0 || index.row() >= zipcodes.count() ||
        index.column() < 0 || index.column() >= MaxColumns)
        return QVariant();
    const ZipcodeItem &item = zipcodes.at(index.row());
```

这个方法(以及 `headerData()` 方法)最与众不同之处是不需要调用对应的基类的方法来处理未想到的情况(而我们必须为那些没有亲自处理的情况返回一个无效的 `QVariant` 变量)。为没有处理到的情况返回任一无效的 `QVariant` 变量以外的值,将导致程序无法工作。

Qt 的模型/视图架构认为 `QAbstractItemModel::data()` 和 `QAbstractItemModel::headerData()` 方法或返回一个可使用的有效的 `QVariant` 变量,或返回一个无效的 `QVariant` 变量(在这种情况下 Qt 尽可能地提供一个它需要的变量)。如果返回一个并不想要的有效的 `QVariant` 变量。例如,如果为没有处理的情况返回一个空字符串或 0 而不是一个无效的 `QVariant` 变量,Qt 就会使用这个的返回值(因为它是一个有效的 `QVariant` 变量)进而出现混乱。

我们从校验模型索引是否有效、行和列是否在有效范围内开始。如果校验通过,则得到一个邮编列表中对应项的只读引用。

我们已经选择了处理两种情况的请求,一种是尺寸发生改变的请求(由 `Qt::SizeHintRole` 指示),另一种是获取项数据的请求(以 `Qt::DisplayRole` 或 `Qt::EditRole` 标志,在本例中我们将它们看成是同义的)。

```
if (role == Qt::SizeHintRole) {
    QStyleOptionComboBox option;
    switch (index.column()) {
        case Zipcode: {
            option.currentText = QString::number(MaxZipcode);
            const QString header = headerData(Zipcode,
                Qt::Horizontal, Qt::DisplayRole).toString();
            if (header.length() > option.currentText.length())
                option.currentText = header;
            break;
        }
        case PostOffice: option.currentText = item.postOffice;
                        break;
        case County: option.currentText = item.county; break;
        case State: option.currentText = item.state; break;
        default: Q_ASSERT(false);
    }
    QFontMetrics fontMetrics(data(index, Qt::FontRole)
        .value<QFont>());
    option.fontMetrics = fontMetrics;
    QSize size(fontMetrics.width(option.currentText),
        fontMetrics.height());
    return qApp->style()->sizeFromContents(QStyle::CT_ComboBox,
        &option, size);
}
```

对于所获取到的对应文本的每一列——除了邮编那列我们总是使用所允许的最大长度或使用该列的标题文本长度两者中较长的。然后创建一个 `fontMetrics` 对象并使用它来计算文本所需要的尺寸。需要注意的是,这里递归调用了 `data()` 方法来获取字体。这里没有亲自处理 `Qt::FontRole` 的情况,因为未处理的情况返回了一个无效的 `QVariant` 变量,所以 Qt 将会替我们进行处理这种情况。

当用户编辑邮编时,会显示一个数字微调框,这需要额外的空间来容纳多出来的微调按钮。类似地,当用户修改州时,则会显示出一个组合框来,这也需要额外的空间来显示下拉按钮。如果不提供这些多余的空间,当用户进行用微调框或组合框进行某一项的编辑时,一些文本内容可能被遮住。对于邮局和县来说,不需要额外的空间,因为它们是通过 `QLineEdit`(仅需要很小的空间来显示边框而已)部件编辑的,但为它们提供额外的空间并没有什么坏处,并能使代码变得更短一些,因为能为所有的列进行相同的计算。

为获取实际所需的尺寸,我们使用了 `QStyleOptionComboBox` 对象,设置它的 `fontMetrics` 成员为对应项字体的 `font metrics` 并且设置它的 `currentText` 成员为对应项的文本内容。然后把这个对象连同文本内容所需的尺寸,作为参数调用 `QStyle::sizeFromContents()` 方法(通过指向全局 `QApplication` 对象的指针 `qApp` 获取应用程序 `QStyle` 的指针)。我们用它来计算在组合框中要显示内容所需的尺寸(在第一个参数中通过 `CT_ComboBox` 进行了指明),并返回该函数返回的大小(这里没有对数字微调框进行同样的处理,因为无论何种情况下,组合框所需要的空间总是比数字设定框要大,所以我们用它来处理这两种情况)。

```
if (role == Qt::DisplayRole || role == Qt::EditRole) {
    switch (index.column()) {
        case Zipcode: return item.zipcode;
        case PostOffice: return item.postOffice;
        case County: return item.county;
        case State: return item.state;
        default: Q_ASSERT(false);
    }
}
```

我们认为显示数据和编辑数据是一样的(相当常见但实际中也并非必须这样)。尽管 `ZipcodeItem` 的成员的数据类型各异(例如 `int` 和 `QString`),但总会返回 `QVariant` 变量。

```
return QVariant();
}
```

对于所有未处理的情况(即没有处理角色的情况),就返回无效的 `QVariant` 对象,并由 Qt 自己来进行处理。

```
QVariant TableModel::headerData(int section,
    Qt::Orientation orientation, int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    if (orientation == Qt::Horizontal) {
        switch (section) {
            case Zipcode: return tr("Zipcode");
            case PostOffice: return tr("Post Office");
            case County: return tr("County");
            case State: return tr("State");
            default: Q_ASSERT(false);
        }
    }
    return section + 1;
}
```

表格视图通常都有行表头(`horizontal header`)和列表头(`vertical header`),所以表格模型应为它们提供标题内容。这里仅处理它们的显示角色(`display role`)——其他角色的请求都将返回一个无效的 `QVariant` 对象。

如果方向是 `Qt::Vertical`,则 `section` 指的是行,如果方向是 `Qt::Horizontal`,则 `section` 指的是列。对于行表头则返回一个合适的文本内容,对于列表头则返回以 1 起计数的行数。

```
int TableModel::rowCount(const QModelIndex &index) const
{
    return index.isValid() ? 0 : zipcodes.count();
}
```

这个方法返回给定的模型索引对应的行数(实际上是子项数)。表格(包括列表)模型项有一个无效的模型索引作为它们的父项,因此如果参数 `index` 是无效的,就返回表格(或列表)中所有的行数。如果行数是固定的(因为还没有实现 `insertRows()` 和 `removeRows()`),就可以使用一个常量来代表行数。

如果参数 `index` 是有效的,即是要返回指定项的子项行数(只对树模型有意义),因此在这种情况下,对于列表和表格模型都必须返回 0。

```
int TableModel::columnCount(const QModelIndex &index) const
{
    return index.isValid() ? 0 : MaxColumns;
}
```

这个方法返回指定模型索引对应的列数。如果参数 `index` 是无效的,就返回表格中的所有列数。(对于列表模型,应从 `QAbstractListModel` 派生,不需要实现这个方法,因为基类提供的就足够了)。在这个示例中,列数是固定的(因为还没有实现 `insertColumns()` 和 `removeColumns()`);但是如果实现了对应的调整列的那些方法,那么列数是可变的。

如果 `index` 是有效的,则是指返回指定的某项的子项列数(几乎没有意义),因此这种情况必须返回 0。

```
bool TableModel::setData(const QModelIndex &index,
                        const QVariant &value, int role)
{
    if (!index.isValid() || role != Qt::EditRole ||
        index.row() < 0 || index.row() >= zipcodes.count() ||
        index.column() < 0 || index.column() >= MaxColumns)
        return false;
    ZipcodeItem &item = zipcodes[index.row()];
    switch (index.column()) {
        case Zipcode: {
            bool ok;
            int zipcode = value.toInt(&ok);
            if (!ok || zipcode < MinZipcode || zipcode > MaxZipcode)
                return false;
            item.zipcode = zipcode;
            break;
        }
        case PostOffice: item.postOffice = value.toString(); break;
        case County: item.county = value.toString(); break;
        case State: item.state = value.toString(); break;
        default: Q_ASSERT(false);
    }
    emit dataChanged(index, index);
    return true;
}
```

除了返回值是代表编辑是否成功的布尔型外,这个方法的开头与 `data()` 函数类似。如果传入的模型索引通过了测试,就创建对应于 `ZipcodeItem` 对象的非常量(可编辑)引用,并且设置对应的列的数据为传入参数 `value` 的值。

我们没有实现字符串列的验证,尽管加入一段代码来拒绝空字符串赋值非常容易(例如, `!(index.column() != Zipcode && value.toString().isEmpty())`)。对于 `zipcode`(邮编),只允许有效的值。

如果未做任何修改,则返回 `false`。相应地,如果编辑成功,则应为发生更改的模型索引发射 `dataChanged()` 信号并且返回 `true`。

我们可能想要编辑过程有一个关联效果,并且模型/视图架构在一定程度通过发射指定矩形区域的左上和右下索引的 `dataChanged()` 信号来支持这种效果。在通常情况下只有一个索引变化,那么我们为两个参数传入同一个索引,就像我们示例中所做的。

```
bool setHeaderData(int, Qt::Orientation, const QVariant&,
                  int=Qt::EditRole) { return false; }
```

我们已经在头文件中实现了这个方法。这里通过返回 `false`(而不管传入的参数是什么)来阻止用户编辑行和列表头。

如果要允许对表头进行编辑则必须发射带有方向和受影响的部分(行或列)的起始点和结束点的 `headerDataChanged()` 信号来实现,并返回 `true`。

```
bool TableModel::insertRows(int row, int count, const QModelIndex&)
{
    beginInsertRows(QModelIndex(), row, row + count - 1);
    for (int i = 0; i < count; ++i)
        zipcodes.insert(row, ZipcodeItem());
    endInsertRows();
    return true;
}
```

一个可删除行的模型必须实现 `insertRows()` 和 `removeRows()` (可删除列的模型则必须实现 `insertColumns()` 和 `removeColumns()`)。如果参数 `row` 为 0,那么新行将插入到所有已有行之前,如果 `row == rowCount()`,那么新行将被追加到最后一行之后。

在结构上,所有的 `insertRows()` 的重新实现都遵循相同模式:在对模型进行任何变更之前先调用 `beginInsertRows()`,然后是执行插入操作的代码,最后在所有对模型的变更都处理完后调用 `endInsertRows()`。如果有任何更改,这个方法必须返回 `true`。

这里的 `beginInsertRows()` 和 `endInsertRows()` 可用于任何列表或表格模型派生类。树模型要稍微复杂些,在下一章中将会看到如何处理。

Zipcodes 示例中,对每一个插入的行插入一个空的 `ZipcodeItem` 对象。事实上 `zipcodes2` 应用程序不直接调用该方法,而是在 `addZipcode()` 函数中调用 `insertRow()` 方法,而 `insertRow()` 方法(在基类中实现)以多态的方式、以 `row` 和 `count` 为 1 作为参数调用 `insertRows()` 方法。

```
bool TableModel::removeRows(int row, int count, const QModelIndex&)
{
    beginRemoveRows(QModelIndex(), row, row + count - 1);
    for (int i = 0; i < count; ++i)
        zipcodes.removeAt(row);
    endRemoveRows();
    return true;
}
```

这个方法是 `insertRows()` 方法的类推,具有相同的结构,只是它调用的是 `beginRemoveRows()` 和 `endRemoveRows()`。这两个调用可以在任何列表或表格模型中使用。

在本例中使用了 `QList::removeAt()` 方法来删除和销毁指定行——这个方法要求参数行号在范围内。需要注意到的一个容易被忽视的地方是,这里总是删除同一行(在删除完一行后所有的后续行都将前移一个位置,所有每一个随后的 `removeAt()` 调用删除的是下一行)。

与 `insertRows()` 类似, `zipcodes2` 应用程序不直接调用 `removeRows()` 方法;而是由 `deleteZipcode()` 方法通过调用基类实现的 `removeRow()` 方法来最终调用以给定行和 `count` 为 1 作为参数的 `removeRows()` 方法。

现在我们介绍完了可编辑、可添加/删除行的表格(或列表)模型所必须重新实现的那些方法。

3.3.2.2 保存与加载表格项的方法

在这一小节,将介绍提供将表格项保存到文件和从文件中加载表格项数据的 `save()` 和 `load()` 方法。它们都会用到我们在前一小节中讨论过的同一个 `zipcode` 邮编文件格式:具有相同的幻数和文件格式版本,使用相同的 `QDataStream` 版本。

```
void TableModel::save(const QString &filename)
{
    if (!filename.isEmpty())
        m_filename = filename;
```



```

    if (m_filename.isEmpty())
        throw AQP::Error(tr("no filename specified"));
    QFile file(m_filename);
    if (!file.open(QIODevice::WriteOnly))
        throw AQP::Error(file.errorString());

    QDataStream out(&file);
    out << MagicNumber << FormatNumber;
    out.setVersion(QDataStream::Qt_4_5);
    QListIterator<ZipcodeItem> i(zipcodes);
    while (i.hasNext())
        out << i.next();
}

```

这个方法的开头与先前介绍的 `StandardTableModel::save()` 方法非常相似,使用一个指定的文件名,或在没有指定新文件名的情况下使用私有成员 `m_filename`。如前所述,先输出幻数和文件格式版本,然后设置数据流的版本。最后使用 `QDataStream::operator <<()` 重载输出所有的项。

```

QDataStream &operator<<(QDataStream &out, const ZipcodeItem &item)
{
    out << static_cast<quint16>(item.zipcode) << item.postOffice
        << item.county << item.state;
    return out;
}

```

在向 `QDataStream` 对象输出整数时,同平时一样,特别指明整型值的符号和位数(16 位无符号整型)是必不可少的。

```

void TableModel::load(const QString &filename)
{
    ...
    QDataStream in(&file);
    quint32 magicNumber;
    in >> magicNumber;
    if (magicNumber != MagicNumber)
        throw AQP::Error(tr("unrecognized file type"));
    quint16 formatVersionNumber;
    in >> formatVersionNumber;
    if (formatVersionNumber > FormatNumber)
        throw AQP::Error(tr("file format version is too new"));
    in.setVersion(QDataStream::Qt_4_5);
    zipcodes.clear();

    ZipcodeItem item;
    while (!in.atEnd()) {
        in >> item;
        zipcodes << item;
    }
    qSort(zipcodes);
    reset();
}

```

这个方法同相对应的 `StandardTableModel::load()` 方法类似。因为文件处理部分与 `save()` 方法相似,所以这里略过这部分内容。后面的代码与 `StandardTableModel::load()` 方法用到的代码一样,用于读取并校验幻数和文件格式版本,接着设置数据流版本。

数据流一旦准备好要读取数据,就会先清除 `Zipcodes` 列表中的老数据,然后使用流的形式读取每一个可用的 `ZipcodeItem` 对象(使用 `QDataStream::operator >>()` 操作符重载)并添加到 `Zipcodes` 列表中。最后对列表按邮编进行排序,并调用 `reset()` 方法将模型数据已变化的信息通知给任何与其相关的视图。

```

QDataStream &operator>>(QDataStream &in, ZipcodeItem &item)
{

```

```
    quint16 zipcode;  
    in >> zipcode >> item.postOffice >> item.county >> item.state;  
    item.zipcode = static_cast<int>(zipcode);  
    return in;  
}
```

使用操作符以流的形式把数据输入到 ZipcodeItem 对象中,而不是单独地读取每个字段值并将它们传入 ZipcodeItem 类的构造函数,这样就将职责区分清楚了。我们能把大部分数据直接以流的形式读入到一个 ZipcodeItem& 的引用中,但是对于整型成员,则必须先把它读入到那个符号和位数均正确的变量(16 位无符号的整型变量)中。

这样,我们就完成了对表格模型的介绍,并且了解了如何去创建一个自定义 QStandardItemModel 派生类来保存表格型数据,以及如何去创建一个自定义 QAbstractTableModel 派生类以提供与其他模型相同的 QAbstractItemModel 应用程序接口。下一章会接着介绍树模型,在它之后的两章则会介绍委托和视图。



第4章 模型/视图树模型

- 用于树的 `QStandardItemModel` 的用法
- 创建自定义树模型

我们在这一章介绍模型/视图中的树模型,如前一章开始时所述,这是以假设读者基本熟悉了 Qt 的模型/视图架构为前提的。

本章将介绍三种树模型。在 4.1 节中将介绍一个使用 `QStandardItem` 派生类来保存数据项的 `QStandardItemModel` 派生类(在前面的章节中我们直接使用了 `QStandardItem`),在 4.2 节我们使用一个自定义模型来替换 `QStandardItemModel`。就像在前一章的表格模型示例中所做的那样,我们将了解如何添加项、如何在项所在位置编辑,以及删除项。在自定义树模型中将实现对项的拖放(drag and drop)、剪切和粘贴(cutting and pasting)、同级移动,提升和降级(promoting and demoting)——在所有的情况下,对项的移动都包含它所有的直接和间接子项。

树模型的工作原理是建立在各项之间父子关系之上,一个项的行号是指在其父项的所有直接子项列表中的位置(从 0 开始)(理论上,一个树模型可以是一个表格形式的递归树,但目前还没有一个 Qt 视图对此进行支持)。

许多树要么有固定的结构,要么有不同类型的项——在这些情况下,对各个项进行移动没有什么实际意义。但对于所有的项(包括子项)都是相同类型的树,任意位置的任意项被移动到树的其他位置的能力就有意义了,我们希望让用户能够对各个项进行自由的移动。对于这个问题,使用将要介绍的自定义 `QAbstractItemModel` 实现起来并不困难。尽管用 `QStandardItemModel::insertRow()` 可以在树中任何地方插入行,`QStandardItemModel::takeRow()` 方法仅仅作用于顶级行,因为它并不接受一个代表父项的 `QModelIndex` 参数,所以我们没有用 `QStandardItemModel` 实现项的移动。这就是说,如果我们想在 `QStandardItemModel` 代表的树中移动行,就不得不做一些烦琐的项复制工作。

在本章的 4.1 节中,我们将了解如何创建一个能够加载和保存定制数据的自定义 `QStandardItemModel` 派生类。每个项由一个自定义的 `QStandardItem` 子类对象表示。在 4.2 节中我们将使用一个自定义的 `QAbstractItemModel` 派生类替换 `QStandardItemModel`,并且用另一个自定义类对象来表示各个项。在 4.2 节的示例中也提供了一些值得注意的辅助功能:在树中移动项(以及它的子项),并且支持拖放。

Timelog 应用程序(`timelog1` 和 `timelog2`)从包含任务的 XML 文件中加载和保存数据。每一个任务都有一个名字,一个完成状态,还有一对或多对开始-结束日期/时间记录。任务名称可以只有简单的字体样式属性——比如粗体、斜体、颜色。各个任务可以任意嵌套,一个指定任务的总时间即是它自己和它的所有子项(包括直接和间接的)的时间之和。在同一时间,只有一个任务可以激活(即正在运行,正在计时中)。

Timelog 应用程序的数据中有一个问题需要注意,单独的开始-结束时间段记录在界面中呈现时并不是独立的。实际上,每一个任务呈现时都是以名称、完成状态和若干关于时间的聚合——任务今日的所有运行时间和其他的关于任务的总体运行时间。这就是说,每一个任务在树中都是以单独的一行来呈现的。

同先前介绍表格模型时相似,对于树也介绍那些类似的内容:如何加载和保存树的项,如何删

除项(包括它所有的直接和间接子项),如何添加、编辑项。另外,对于使用自定义模型的 `timelog2` 应用程序,我们还会了解到如何移动任务(包括它所有的直接和间接的子项),并提供通过键盘、菜单项、工具按钮和拖放的方式使用这些功能。此外,还将了解到如何隐藏和显示任务——对于 `Timelog` 示例来说这基于每个项的完成状态。但我们将尽力避免涉及与模型/视图无关的那些功能——尤其是那些与开始、停止计时以及计时图标动画相关的代码。

两个应用程序都使用了一个自定义的富文本(rich text)列委托对象来处理任务名称的显示和编辑。这些将在第 5 章进行介绍。

4.1 用于树 `QStandardItemModel` 的用法

`timelog1` 示例使用了一个 `QStandardItemModel` 子类来加载、编辑和保存数据,并且用了一个带有自定义委托对象的 `QTreeView` 来显示和编辑数据。用户界面是非常传统的形式,有一个菜单栏、若干菜单项和工具栏。像往常一样,我们关注于模型/视图方面的细节,而略过大多数窗口部件的创建和布局以及许多方法。

如图 4.1 所示,应该能从界面图中看出来一些任务名称使用了多种字体样式和颜色。在 `timelog1` 示例中,任务在树中的位置在添加进树时就固定了下来——在下一节中的 `timelog2` 示例里面就会了解如何支持任务(包括所有直接和间接子项)的任意移动。

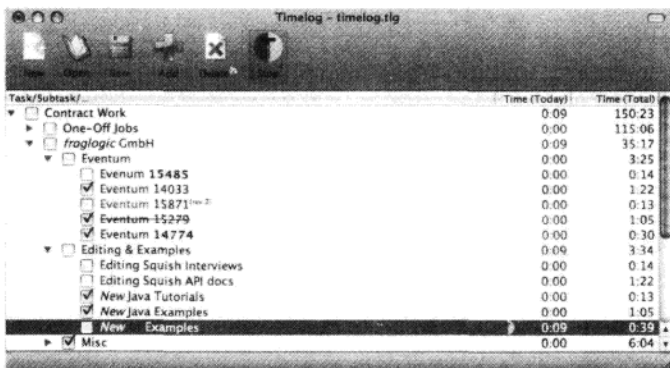


图 4.1 `timelog1` 应用程序示例

`timelog1` 示例的用户界面支持传统的以文档为中心的功能,即创建新文件、打开已存在文件、保存到文件(文件保存了自定义 XML 格式的任务树),还包括添加新项和删除已存在的项。另外用户还可以开始或结束一个任务的计时(会添加一个新的开始-结束时间段到任务的时间列表中),以及隐藏或显示“已完成”(复选框被选中的)的任务。

4.1.1 通过用户界面改变树模型

在这个小节中将介绍该应用程序的梗概以及用户界面,以使我们有足够的信息来理解随后一小节中与模型相关的讨论和代码片段。首先从主窗口类定义中提取的一部分代码开始介绍,然后来看看主窗口的构造函数,包括创建树模型和树部件以及最重要的信号-槽连接。其他的我们想要了解的方法和那些添加、删除任务,隐藏(或显示)“已完成”(选中复选框)任务的方法(用来操作树结构的方法,也就是说,移动任务的方法,仅在自定义模型的版本中提供——将在下一节进行介绍)。

```

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent=0);

public slots:
    void stopTiming();

protected:
    void closeEvent(QCloseEvent*);

private slots:
    void fileNew();
    void fileOpen();
    bool fileSave();
    bool fileSaveAs();
    void editAdd();
    void editDelete();
    void editHideOrShowDoneTasks(bool hide);
    void setDirty(bool dirty=true) { setWindowModified(dirty); }
    void load(const QString &filename,
              const QStringList &taskPath=QStringList());

private:
    ...
    QTreeView *treeView;
    StandardTreeModel *model;
};

```

主窗口类并没有什么令人惊喜的地方。这里略过了几个私有方法和一些私有数据成员,在进行函数的说明涉及到这些内容时,在必要的情况下再做介绍。这里没有列出来文件处理的方法,但在接下来讨论从构造函数中摘出来的三部分内容时,将讨论 load() 方法。

```

const QString FilenameSetting("Filename");
const QString GeometrySetting("Geometry");
const QString CurrentTaskPathSetting("CurrentTaskPath");
const int FirstFrame = 0;
const int LastFrame = 4;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    createModelAndView();
    createActions();
    createMenusAndToolBar();
    createConnections();
    AQP::accelerateMenu(menuBar());
    setWindowTitle(tr("%1 (QStandardItemModel)[*]")
                  .arg(QApplication::applicationName()));
}

```

构造函数依惯例开始,创建模型和视图,然后创建动作、菜单和工具栏,接着创建信号-槽连接。这里将略过创建动作、菜单和工具栏的内容,因为这些内容我们都比较熟悉且与我们关注的模型/视图开发不相关。

```

timer.setInterval(333);
iconTimeLine.setDuration(5000);
iconTimeLine.setFrameRange(FirstFrame, LastFrame + 1);
iconTimeLine.setLoopCount(0);
iconTimeLine.setCurveShape(QTimeLine::LinearCurve);

```

显示上面的这段构造函数中的代码是为了提供上下文信息。应用程序使用了两个定时器,类型为 QTimer 的定时器 timer,用于保持项的计时统计是最新的, QTimeLine 类型的 iconTimeLine 则用于产生一个正在计时的任务的动画图标。timer 每 1/3 s 更新一次,iconTimeLine 以每 5 s 走 5 帧的速度进行重复循环(指定循环次数为 0 即可实现无限循环)。曲线形状决定了在两个帧之间的时间

间隔: `QTimeLine::LinearCurve` 用来确保相同的时间间隔 (Qt 4.6 已经引入了一个比使用定时器或 `QTimeLine` 更高级、先进的新的动画框架,将在第 13 章进行介绍)。

```
QSettings settings;
restoreGeometry(settings.value(GeometrySetting).toByteArray());
QString filename = settings.value(FilenameSetting).toString();
if (filename.isEmpty())
    QTimer::singleShot(0, this, SLOT(fileNew()));
else
    QMetaObject::invokeMethod(this, "load", Qt::QueuedConnection,
        Q_ARG(QString, filename),
        Q_ARG(QStringList, settings.value(
            CurrentTaskPathSetting).toStringList()));
}
```

在构造函数末尾,试图加载上次使用过的任务文件,或者创建一个新任务文件以便于用户添加任务。我们的策略是,总是在主窗口创建后加载文件,以使主窗口尽可能快地显示出来,即使被加载的文件非常大(这还有另一个优点就是文件只会在主窗口完全创建后才开始加载)。因为需要给槽传递参数,所以不能使用一个只执行一次的定时器 (single shot timer) 来载入文件。

`QMetaObject::invokeMethod()` 用来通过事件队列来调用一个槽,这就是说,把调用放入到一个队列中,在队列为空时就会调用指定的槽,在这里就是在构造函数执行完毕之后进行调用。第一个参数是接收对象,第二个参数是要调用的槽的名字,第三个参数是连接类型,剩下的是槽的参数表 (事实上 `QMetaObject::invokeMethod()` 也可以通过使用连接类型为 `Qt::DirectConnection` 来立即调用一个槽——对于由用户直接或间接选择方法和参数进行的调用,比如通过一个对话框,这种方法可能比较有用。在第 12 章将介绍一个 `Qt::DirectConnection` 的示例)。

传入到槽 `load()` 中的两个参数是任务文件名和任务路径。任务路径是由各个任务文本字符串组成的字符串列表对象,标志一个特定的任务。举个例子,在图 4.1 中,高亮显示项的任务路径是 `["Contract Work", "<i>froglogic</i> GmbH", "Editing & Examples", "<i>New</i> Qt Examples"]`。任务文本使用一个简单的 HTML 子集来表示字体效果,在第 5 章介绍富文本委托对象的时候将进行介绍。下面显示的是从 `load()` 方法中摘取出来的代码,用于显示正在使用的任务路径:

```
model->load(filename);
if (!taskPath.isEmpty()) {
    if (QStandardItem *item = model->itemForPath(taskPath))
        setCurrentIndex(item->index());
}
```

这段代码放在了 `try...catch` 代码块中以防止加载失败。`load()` 方法既被构造函数用来恢复上次使用过的文件,也被 `fileOpen()` 函数使用——在这种情况下,任务路径是一个空字符串列表。`QStandardItem::index()` 方法返回了一个项的模型索引。自定义方法 `StandardItemModel::itemForPath()` (以及对应的 `StandardItemModel::pathForIndex()` 方法)将在后面介绍。

```
void MainWindow::setCurrentIndex(const QModelIndex &index)
{
    if (index.isValid()) {
        treeView->scrollTo(index);
        treeView->setCurrentIndex(index);
    }
}
```

为了方便起见,我们创建了这个辅助方法,因为不止一处要用到这个功能。它确保指定的模型索引对应的项在视图中是可见的并且被选中,如果有必要,那么还会滚动视图或展开项。

```
void MainWindow::createModelAndView()
{
    model = new StandardTreeModel(this);
    treeView->setAllColumnsShowFocus(true);
    treeView->setItemDelegateForColumn(0, new RichTextDelegate);
    treeView->setModel(model);
    setCentralWidget(treeView);
}
```

这里创建了一个 `StandardTreeModel` (一个 `QStandardItemModel` 子类) 实例, 并使用一个标准的 `QTreeView` 部件来显示数据。

这个树形视图显示了三列——任务名称, 今日的任务执行时间和总任务执行时间。我们留下了时间列让 `QTreeView` 的内置委托对象来处理显示的问题, 但是对于任务列, 就必须使用一个自定义的委托对象以 HTML 形式来显示内容, 而不是用普通文本形式来显示。使用自定义列委托对象的一大优点, 就是在大多数情况下一个特定列的所有项通常会使用相同的数据类型, 因此就能创建专为某种数据使用的列委托对象, 而且极有可能比模型专用的委托对象更具有复用性(委托将在下一章介绍)。

尽管没有进行介绍, 但如果模型测试模块(model test module)可用的话, 在源代码中已经使用 `#ifdef` 编译预处理创建了一个 `ModelTest` 对象, 就像在先前的 Zipcodes 示例中所做的那样。尽管模型是一个仅仅添加载入了和保存数据功能的 `QStandardItemModel` 派生类, 且并没有改变内置功能的行为, 它仍然会导致模块测试(版本 0.2)弹出断言(assert)。实际上有两个模型测试强调的问题。第一个(modeltest.cpp 中的第 106 行)看上去像一个在 `QStandardItemModel` 中无害的错误(无效索引对应的 flag 不是 0)^①。第二个仿佛是由于热心过度而不是一个真正的问题(在代码中的 341 行之上有一个注释, 暗示了失败的断言可以安全地注释掉)。在这两种情形下把有疑问的行注释掉, 因为这两个看起来像是假警报。

```
void MainWindow::createConnections()
{
    connect(treeView->selectionModel(),
           SIGNAL(currentChanged(const QModelIndex&,
                                const QModelIndex&)),
           this, SLOT(updateUi()));
    connect(model, SIGNAL(itemChanged(QStandardItem*)),
           this, SLOT(setDirty()));
    connect(model, SIGNAL(rowsRemoved(const QModelIndex&,int,int)),
           this, SLOT(setDirty()));
    connect(model, SIGNAL(modelReset()), this, SLOT(setDirty()));
}
```

这里仅显示了前面的几个连接。槽 `updateUi()` (这里没有介绍) 根据应用程序的状态, 通过调用可用/禁用的动作来保持用户界面为最新状态。从模型发射出来的所有(间接的)信号连接都会被设置 `windowModified` 属性, 以使得用户有机会来保存还未保存的修改。

大部分其他连接是简单地把一个动作的 `triggered()` 信号连接到对应的槽中——例如连接 `fileNewAction` 到 `fileNew()`, 连接 `editAddAction` 到 `editAdd()`。也有两三个与定时器相关的连接, 用于计时项以及显示计时项的动画图标。

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    stopTiming();
    if (okToClearData()) {
        QSettings settings;
        settings.setValue(GeometrySetting, saveGeometry());
        settings.setValue(FilenameSetting, model->filename());
    }
}
```

^① 行号在这次写作中已经进行了更正(subversion 中的版本与打包的版本不一致), 但仍可能与下载的模式测试不一致。

```

        settings.setValue(CurrentTaskPathSetting,
            model->pathForIndex(treeView->currentIndex()));
        event->accept();
    }
    else
        event->ignore();
}

```

如果用户退出应用程序, `closeEvent()` 事件处理函数将被调用。 `stopTiming()` 方法(这里没有介绍)如其名称所示即会停止计时项的计时(停止正在运行的任务)。 `setDirty()` 方法被用来确保窗口的修改状态正确反映了是否有还未保存的变化, 该状态用 `okToClearData()` 方法来进行检查。如果 `okToClearData()` 方法返回 `true`, 就保存用户的设置(主窗口的位置和大小、任务文件、当前选中任务的任務路径), 然后接受关闭事件以允许应用程序退出。为完整起见, 我们看一下 `okToClearData()` 方法以及它依赖的 `AQP::okToClearData()` 函数。

```

bool MainWindow::okToClearData()
{
    if (isWindowModified())
        return AQP::okToClearData(&MainWindow::fileSave, this,
            tr("Unsaved changes"), tr("Save unsaved changes?"));
    return true;
}

```

既然向用户提供一个 `save unsaved changes` 对话框是一个共同的需求, 这里创建了一个辅助函数 `AQP::okToClearData()` 来提供所需要的功能。这个函数接受了一个函数指针, 如果用户请求保存, 那么就指向执行保存的方法。另一个参数是对话框应该显示在它之上的窗口指针(本例即为主窗口), 也是调用保存方法的实例。还接受标题栏、显示的文本内容和一些其他可选内容(在本例中没有传递该参数)。传入的函数指针必须是第二个参数所指的窗口的一个成员方法, 并且这个方法最后要返回一个布尔值来表明成功与否。传入一个方法的指针的语法是 `&ClassName::MethodName`。

```

template<typename T>
bool okToClearData(bool (T::*saveData)(), T *parent,
    const QString &title, const QString &text,
    const QString &detailedText=QString())
{
    QScopedPointer<QMessageBox> messageBox(new QMessageBox(parent));
    messageBox->setWindowModality(Qt::WindowModal);
    messageBox->setIcon(QMessageBox::Question);
    messageBox->setWindowTitle(QString("%1 - %2")
        .arg(QApplication::applicationName()).arg(title));
    messageBox->setText(text);
    if (!detailedText.isEmpty())
        messageBox->setInformativeText(detailedText);
    messageBox->addButton(QMessageBox::Save);
    messageBox->addButton(QMessageBox::Discard);
    messageBox->addButton(QMessageBox::Cancel);
    messageBox->setDefaultButton(QMessageBox::Save);
    messageBox->exec();
    if (messageBox->clickedButton() ==
        messageBox->button(QMessageBox::Cancel))
        return false;
    if (messageBox->clickedButton() ==
        messageBox->button(QMessageBox::Save))
        return (parent->*saveData)();
    return true;
}

```

指定一个无参数的成员函数作为参数的语法形如 `returnType (Type::method)()`, `returnType` 是该

方法的返回值类型(有可能是 void), Type 是该方法所属的类(例如 MainWindow), method 是在该函数(在这里是 okToClearData)中给出的方法名字——它可以是任何名称,特别是它可以与真实名称不同(然而,我们调用 okToClearData() 时,必须明确地给出类和方法——比如 &MainWindow::fileSave, 就像先前看到的一样)。

这里选择把 okToClearData() 做成一个模板函数,这样就不需要把类名写死(hard-code)了。这意味着我们能使用任何返回布尔值的方法,比如 MainWindow::save()。不用管它的真实名称是什么,在这个函数中该方法被称为 saveData()。

在题为“避免使用 Qt 的静态便利 QMessageBox 函数”的阴影部分和题为“Qt 的智能指针”的阴影部分中分别讲述了一个 QMessageBox 对象的创建过程以及 Qt 4.6 中的 QScopedPointer^①。

如果用户选择保存,就调用参数所指的方法,然后返回该方法的返回值。调用一个以指针传入的无参数的函数的语法为(object -> * method)()。所以在这个示例中,实际上调用的是 MainWindow::fileSave()。如果文件没有命名并且用户在另存为对话框中点击了取消按钮(或者通过对话框关闭按钮进行取消等),fileSaveAs()方法(从 fileSave()方法中调用的)将返回 false,相应的该返回值将被返回,在这种情况下 okToClearData()将返回 false,所以尽管没有保存更改,也不会丢失什么更改。只有在一种情况下用户可以避免保存这些更改,即当用户有一个未命名(也就是新建文件)文件并且明确地选择放弃那些修改时。

在 timelog1 示例中,当任务创建后(这个约束对 timelog2 示例无效,在下一节将会介绍到)每一个任务在树中的位置是固定的。因此在添加一个新任务时,必须给用户指定新任务应该放在哪里的选项。这是在 editAdd()方法中处理的,这里把它分成三部分进行介绍。

```
void MainWindow::editAdd()
{
    QModelIndex index = treeView->currentIndex();
    StandardTreeModel::Insert insert = StandardTreeModel::AtTopLevel;
```

首先获取当前项的模型索引(如果没有当前项的话,那它就是无效的——比如用户正好选择了 File→New 菜单)。我们仍然做了一个初始的假设,假设新项必须添加为一个使用 StandardTreeModel 类中的一个枚举值的顶级项——如果没有现存项,这是唯一的选择。

```
if (index.isValid()) {
    QStandardItem *item = model->itemFromIndex(index);
    QScopedPointer<QMessageBox> messageBox(new QMessageBox(this));
    messageBox->setWindowModality(Qt::WindowModal);
    messageBox->setIcon(QMessageBox::Question);
    messageBox->setWindowTitle(tr("%1 - Add Task")
        .arg(QApplication::applicationName()));
    messageBox->setText(tr("<p>Add at the top level or as a "
        "sibling or child of\n'%1'?").arg(item->text()));
    messageBox->addButton(tr("&Top Level"),
        QMessageBox::AcceptRole);
    QAbstractButton *siblingButton = messageBox->addButton(
        tr("&Sibling"), QMessageBox::AcceptRole);
    QAbstractButton *childButton = messageBox->addButton(
        tr("&Child"), QMessageBox::AcceptRole);
    messageBox->setDefaultButton(
        qobject_cast<QPushButton*>(childButton));
    messageBox->addButton(QMessageBox::Cancel);
    messageBox->exec();
    if (messageBox->clickedButton() ==
```

① 源代码中我们使用了 #if QT_VERSION 编译预处理,以使得代码能够在 Qt 4.5 中使用 QSharedPointer 进行编译。

```

        messageBox->button(QMessageBox::Cancel))
        return;
    if (messageBox->clickedButton() == childButton)
        insert = StandardTreeModel::AsChild;
    else if (messageBox->clickedButton() == siblingButton)
        insert = StandardTreeModel::AsSibling;
}

```

如果有一个当前项,就给用户一个选择把这个新项添加为顶级项,或当前项的同级项或子项的机会。如果用户不取消并且不选择顶级项,就改变 insert 变量为相应的值。

要注意到这里我们设置信息框的信息内容是以 <p> (HTML 中的段落起始标签) 开头的。这就确保了信息文本被解释成 HTML 形式并正确地显示出来。这一点很重要,因为信息内容包含了当前任务的名称,而这个名称可能含有 HTML 标记。

```

    if (QStandardItem *item = model->insertNewTask(insert,
        tr("New Task"), index)) {
        QModelIndex index = item->index();
        setCurrentIndex(index);
        treeView->edit(index);
        setDirty();
        updateUi();
    }
}

```

这里向模型请求在指定位置插入一个新任务,指定了默认文本和给定的父项(当插入的是顶级项时,忽略这个参数)。如果插入成功(如预期的那样),就获取新插入项的模型索引,设置它为当前项,并且发起编辑,以使用户能够自己输入名称以取代默认的 New Task。

```

void MainWindow::editDelete()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;
    QStandardItem *item = model->itemFromIndex(index);
    if (item == timedItem)
        stopTiming();
    QString name = item->text();
    int rows = item->rowCount();
    QString message;
    if (rows == 0)
        message = tr("<p>Delete '%1'").arg(name);
    else if (rows == 1)
        message = tr("<p>Delete '%1' and its child (and "
            "grandchildren etc.)").arg(name);
    else if (rows > 1)
        message = tr("<p>Delete '%1' and its %2 children (and "
            "grandchildren etc.)").arg(name).arg(rows);
    if (!AQP::okToDelete(this, tr("Delete"), message))
        return;
    model->removeRow(index.row(), index.parent());
    setDirty();
    updateUi();
}

```

如果当前没有任何选中项,那么该方法什么也不做就返回。否则它向模型请求获取当前选中项的索引对应的项,以及对应的文本和直接子项数量。这里使用了三个单独的字符串来分别代表三种情况(没有子项、只有一个直接子项、有多个直接子项),这能够使翻译者的工作变得容易些。

随后提示用户进行删除确认,如果用户点击了 Delete 按钮,就通知模型执行删除操作(在前面已经介绍过了 AQP::okToDelete() 方法)。在最终的基本类 QAbstractItemModel 中,removeRow() 方法什么

也没有做并且返回了 `false`,但在我们的 `StandardItemModel` 类的直接基类 `QStandardItemModel` 中有它的重新实现,它正确地实现了删除指定行(以及它的所有直接和间接子项)并返回 `true`。

```
void MainWindow::editHideOrShowDoneTasks(bool hide)
{
    hideOrShowDoneTask(hide, model->invisibleRootItem());
}

void MainWindow::hideOrShowDoneTask(bool hide, QStandardItem *item)
{
    QModelIndex index = item->parent() ? item->parent()->index()
                                         : QModelIndex();
    bool hideThisOne = hide && (item->checkState() == Qt::Checked);
    treeView->setRowHidden(item->row(), index, hideThisOne);
    if (!hideThisOne) {
        for (int row = 0; row < item->rowCount(); ++row)
            hideOrShowDoneTask(hide, item->child(row, 0));
    }
}
```

`editHideOrShowDoneTasksAction` 是一个连接到 `editHideOrShowDoneTasks()` 槽的切换动作(toggle action),该槽会触发对 `hideOrShowDoneTask()` 方法的递归调用。

在 `hideOrShowDoneTask()` 方法中,首先判断当前项是否应该隐藏,然后调用 `QTreeView::setRowHidden()` 相应的项来隐藏或显示当前项对应的行。如果对应的行已经是隐藏的,那么我们不需要担心任何子项,因为它们已经自动隐藏了;但是如果当前项没有被隐藏,就必须递归地检查该项的每个子项,酌情进行隐藏(或显示)。

现在已经介绍了 `Timelog` 示例的用户界面中与使用树模型相关的基本要素。这将给我们足够的信息来理解在下一节中将要介绍的自定义 `QStandardItem` 子类,以及在之后一小节中将要介绍的 `QStandardItemModel` 子类。

4.1.2 用于树中各项的 `QStandardItem` 派生类

对于 `timelog1` 应用程序,选择了使用一个自定义的 `QStandardItem` 派生类,因为我们想添加一些与任务的开始-结束时间段有关的数据成员和方法到类中。下面是这个类的完整定义:

```
class StandardItem : public QStandardItem
{
public:
    explicit StandardItem(const QString &text, bool done);
    QStandardItem *todayItem() const { return m_today; }
    QStandardItem *totalItem() const { return m_total; }
    void addDateTime(const QDateTime &start, const QDateTime &end)
    { m_dateTimes << qMakePair(start, end); }
    QList<QPair<QDateTime, QDateTime>> dateTimes() const
    { return m_dateTimes; }
    void incrementLastEndTime(int msec);

    QString todaysTime() const;
    QString totalTime() const;

private:
    int minutesForTask(bool onlyForToday) const;
    QStandardItem *m_today;
    QStandardItem *m_total;
    QList<QPair<QDateTime, QDateTime>> m_dateTimes;
};
```

不同寻常地,这里使用了两个 `QStandardItem` 类型的指针来显示任务今天的执行时间和总共的执行时间。为了方便使用而牺牲了一点点内存。这里使用了一个 `QList` 对象来保存所有的任务开始-结束时间段,当用户开始为任务计时时,把数据添加到这个列表中。

这里唯一要介绍的方法是构造函数;其他的非内联(non-inline)方法用于当任务进行计时的时候累加时间、计算次数,并且把时间以字符串或日期时间型的形式返回,因此它们与模型/视图编程无关,但是可以在本书的源代码中找到它们。

```
StandardItem::StandardItem(const QString &text, bool done)
    : QStandardItem(text)
{
    setCheckable(true);
    setCheckState(done ? Qt::Checked : Qt::Unchecked);
    setFlags(Qt::ItemIsSelectable|Qt::ItemIsEnabled|
        Qt::ItemIsEditable|Qt::ItemIsUserCheckable);
    m_today = new QStandardItem;
    m_today->setFlags(Qt::ItemIsSelectable|Qt::ItemIsEnabled);
    m_today->setTextAlignment(Qt::AlignVCenter|Qt::AlignRight);
    m_total = new QStandardItem;
    m_total->setFlags(Qt::ItemIsSelectable|Qt::ItemIsEnabled);
    m_total->setTextAlignment(Qt::AlignVCenter|Qt::AlignRight);
}
```

这里设置了任务项为可复选的(用以表示“完成”状态),并设置它为可用的、可选中的、可编辑的并且可复选的(checkable),这样用户就可以通过设置复选框为选中或未选中来切换任务是否完成状态——用户通过鼠标点击或在选中该项时按下空格键即可进行复选状态的切换。

在 StandardTreeModel 派生类中,我们仅仅创建了若干个 StandardItem 项,每一个项代表一个任务。这每一项又相应地创建了若干个 QStandardItem 来用于任务时间的表示。对于那些时间项来说,我们只允许它是可选中的、可用的,因为我们并不希望用户能够编辑它们,也不希望它们有复选框。

现在我们知道了 StandardItem 项中所包含的数据成员以及它提供的方法,现在是来看看包含了所有任务的 StandardTreeModel 类的时候了。

4.1.3 用于树的 QStandardItemModel 派生类

StandardTreeModel 是一个用来代表包含任务的树的 QStandardItemModel 派生类。除了提供文件处理功能(load()和 save())之外,这个类还提供了在 MainWindow::editAdd()方法中所见到的 insertNewTask()方法(用于添加一个新任务)。它还有 pathForIndex()和 itemForPath()方法,用来处理在前面讨论过的任务路径,将在这一小节对它们进行介绍。这个类的唯一的自定义私有成员是一个 QString 类型的文件名。

应用程序的数据保存在磁盘上的一个 XML 格式的文件中。如图 4.2 所示的是这样的一个文件的一部分内容。

```
<TASK NAME="Editing & Examples" DONE="0">
  <WHEN START="2009-03-18T13:23:59" END="2009-03-18T13:25:15"/>
  <TASK NAME="Editing Squish Interviews" DONE="0">
    <WHEN START="2009-02-03T16:19:35" END="2009-02-03T16:34:02"/>
  </TASK>
  <TASK NAME="Editing Squish API docs" DONE="0">
    <WHEN START="2008-11-14T07:46:31" END="2008-11-14T08:25:55"/>
    <WHEN START="2008-11-14T09:55:26" END="2008-11-14T10:21:11"/>
    <WHEN START="2009-02-25T10:50:20" END="2009-02-25T10:51:29"/>
    <WHEN START="2009-02-25T10:52:02" END="2009-02-25T11:09:33"/>
  </TASK>
</TASK>
...
```

图 4.2 Timelog 数据文件的一部分

每个任务的名称和完成状态都存储在 TASK 标签 (tag) 的属性中。任务名称能够包含 HTML 标记 (markup), 但必须是已经正确地进行过转义处理的, 这样它们才不会与 XML 标记相冲突。完成状态属性用“0”表示 false, 用“1”表示 true。那些开始-结束时间段记录存储在对应的 TASK 标签内的 WHEN 标签的属性中, 是 ISO 8601 标准格式的日期、时间字符串。任务的层级是很自然地通过在 TASK 标签间任意层数的嵌套来实现的。

在这一小节中, 将查看所有的 StandardTreeModel 的方法, 因为它们都与模型/视图编程相关。

```
StandardTreeModel::StandardTreeModel(QObject *parent)
    : QStandardItemModel(parent)
{
    initialize();
}

void StandardTreeModel::initialize()
{
    setHorizontalHeaderLabels(QStringList() << tr("Task/Subtask/...")
        << tr("Time (Today)") << tr("Time (Total)"));
    for (int column = 1; column < columnCount(); ++column)
        horizontalHeaderItem(column)->setTextAlignment(
            Qt::AlignVCenter|Qt::AlignRight);
}
```

这里把初始化过程单独放在一个函数中, 这是因为它会在两个不同的地方被调用。

```
void StandardTreeModel::clear()
{
    QStandardItemModel::clear();
    initialize();
}
```

基类的 clear() 方法不仅除去了所有的模型项, 还把表头也去除了。因此在清除完后调用 initialize() 方法重新创建表头。

```
void StandardTreeModel::save(const QString &filename)
{
    if (!filename.isEmpty())
        m_filename = filename;
    if (m_filename.isEmpty())
        throw AQP::Error(tr("no filename specified"));
    QFile file(m_filename);
    if (!file.open(QIODevice::WriteOnly|QIODevice::Text))
        throw AQP::Error(file.errorString());

    QXmlStreamWriter writer(&file);
    writer.setAutoFormatting(true);
    writer.writeStartDocument();
    writer.writeStartElement("TIMELOG");
    writer.writeAttribute("VERSION", "2.0");
    writeTaskAndChildren(&writer, invisibleRootItem());
    writer.writeEndElement(); // TIMELOG
    writer.writeEndDocument();
}
```

这个方法的开始部分, 代码与 Zipcodes 示例中相似, 如果给出了新文件名, 就使用给定的新文件名, 否则使用已存在的文件名。调用者期望能捕获和处理任何异常。AQP::Error 类与在前面看到的内容相同。

QXmlStreamWriter 类不需要进行子类化就能输出我们想要的内容。如果 autoFormatting 属性为 true, 那么将以缩进和添加新行的有利于阅读的格式输出 XML 内容; 否则它将以紧凑的格式去除任何不必要的空白字符来进行输出。writeStartDocument() 调用时将在文件头输出 "<? xml version = \"1.0\" encoding = \"UTF-8\"? >\"。当然, 如果调用 QXmlStreamWriter::setCodec() 函数应用了我们想要的编码, 那么 encoding 属性的值 (包括它实际使用到的内部编码) 将发生改变。

我们在 TIMELOG 标签(没有显示出来)中关闭各层级的 TASK 标签,并为 TIMELOG 标签添加了一个 VERSION 属性以便于在将来进行文件格式变更。QXmlStreamWriter::writeAttribute()方法接受一个属性名称和一个值——不需要做任何 XML 转义操作,因为 QXmlStreamWriter 会自动处理的。然后调用 writeTaskAndChildren()方法来输出所有的任务。

```
const QString TaskTag("TASK");
const QString NameAttribute("NAME");
const QString DoneAttribute("DONE");
const QString WhenTag("WHEN");
const QString StartAttribute("START");
const QString EndAttribute("END");

void StandardTreeModel::writeTaskAndChildren(QXmlStreamWriter *writer,
                                             QStandardItem *root)
{
    if (root != invisibleRootItem()) {
        StandardItem *item = static_cast<StandardItem*>(root);
        writer->writeStartElement(TaskTag);
        writer->writeAttribute(NameAttribute, item->text());
        writer->writeAttribute(DoneAttribute,
                               item->checkState() == Qt::Checked ? "1" : "0");
        QListIterator<
            QPair<QDateTime, QDateTime> > i(item->dateTimes());
        while (i.hasNext()) {
            const QPair<QDateTime, QDateTime> &dateTime = i.next();
            writer->writeStartElement(WhenTag);
            writer->writeAttribute(StartAttribute,
                                   dateTime.first.toString(Qt::ISODate));
            writer->writeAttribute(EndAttribute,
                                   dateTime.second.toString(Qt::ISODate));
            writer->writeEndElement(); // WHEN
        }
    }
    for (int row = 0; row < root->rowCount(); ++row)
        writeTaskAndChildren(writer, root->child(row, 0));
    if (root != invisibleRootItem())
        writer->writeEndElement(); // TASK
}
```

注意这里并没有在标签和属性名称中使用 tr() 函数,这是因为它们是文件格式的一部分而且严格来说不是专为人来阅读的。

这个方法输出了一个任务及其所有子任务(递归的),但是跳过了不可见的根项(root item)。首先输出具有名称和完成属性的 TASK 标签,然后遍历所有任务的开始-结束时间段,对每一个时间段都输出一个 WHEN 标签。然后输出该任务的所有子任务(依次输出它的子任务,诸如此类)。这种递归的结构能正确的保留下来,因为每一个任务只有在输出完所有的子任务(及子项的子项,等等)后才会输出 TASK 结束标签。

现在我们已经了解了任务是如何被保存的,下面看一下它们是如何被加载的。我们把 load() 方法分成两部分以易于讲解。

```
void StandardTreeModel::load(const QString &filename)
{
    ...
    clear();

    QStack<StandardItem*> stack;
    stack.push(invisibleRootItem());
    QXmlStreamReader reader(&file);
    while (!reader.atEnd()) {
        reader.readNext();
        if (reader.isStartElement()) {
```

```

    if (reader.name() == TaskTag) {
        const QString name = reader.attributes()
            .value(NameAttribute).toString();
        bool done = reader.attributes().value(DoneAttribute)
            == "1";
        StandardItem *nameItem = createNewTask(stack.top(),
            name, done);
        stack.push(nameItem);
    }
    else if (reader.name() == WhenTag) {
        const QDateTime start = QDateTime::fromString(
            reader.attributes().value(StartAttribute)
                .toString(), Qt::ISODate);
        const QDateTime end = QDateTime::fromString(
            reader.attributes().value(EndAttribute)
                .toString(), Qt::ISODate);
        StandardItem *nameItem = static_cast<StandardItem*>(
            stack.top());
        nameItem->addDateTime(start, end);
    }
}
else if (reader.isEndElement()) {
    if (reader.name() == TaskTag)
        stack.pop();
}
}
}

```

load()方法的开始部分几乎是以与 save()方法相同的方式来处理文件名称的,这里略过不提。唯一不同的是这里使用了 QIODevice::ReadOnly 标志以二进制只读模式打开文件。QXmlStreamReader 将读取 <?xml?> 标签来判断要使用的正确的字符编码,如果没有指定的话,就默认为 UTF-8。

一旦文件成功打开,先清除现有的项,以便把 XML 文件中读取的数据添入模型中。

不需要子类化 QXmlStreamReader,这是因为这个类提供的功能足够我们直接用了。

创建任何一个新任务时都必须给它指定一个父项(可以是任务的直接父项,或是基类提供的不可见的作为所有顶级项的父项的根项)。这里使用 QStack<QStandardItem*> 来保存父项,把任务压到栈中并且在必要的时候把它们弹出来。

当遇到一个 TASK 的开始标签时,创建一个新的 StandardItem 对象来代表一个任务,从栈中取最顶端的那个项作为新建任务的父项,任务名称和完成状态则从标签的属性中获取。一旦新任务创建出来,就把它压到栈的顶端。

QXmlStreamReader::attributes() 方法把当前元素的属性作为一个 QXmlStreamAttributes 的对象返回。QXmlStreamAttributes::value() 方法根据传入的属性名来返回对应的 QStringRef 类型的属性值,并且是未进行 XML 转义的。举个例子,任务名称“Editing & Examples”在 XML 文件中保存成“Editing & Examples”,但通过 value() 返回的则是它原本的形式。

尽管能容易的使用标准的比较操作符来在 QString 和 QStringRef 对象之间进行比较,但如果确实需要 QStringRef 的文本内容,那么必须调用 toString() 方法来获取。例如,把一个任务的名字通过 QStringRef::toString() 方法放到一个 QString 对象中,但是对于完成状态标志,仅需拿属性值与“1”比较一下以确定任务是否完成就可以了。

当遇到一个 WHEN 开始标签时,把开始-结束时间段提取出来,然后把它们压入到保存日期时间列表的栈中。每当遇到 TASK 结束标签时,就弹出栈顶。

```

    if (reader.hasError())
        throw AQP::Error(reader.errorString());
    if (stack.count() != 1 || stack.top() != invisibleRootItem())
        throw AQP::Error(tr("loading error: possibly corrupt file"));
    calculateTotalsFor(invisibleRootItem());
}

```

如果 XML 解析发生错误时, `QXmlStreamReader::atEnd()` 函数将返回 `true` (这将会终止循环), 并且 `QXmlStreamReader::hasError()` 将返回 `true` 以告知我们发生错误了。如果栈中不是刚好只剩下末尾的一项任务 (即不可见的根项), 就表明有地方出错了。只要发生上述两者中的任何一种错误, 就会抛出一个异常留给调用者来处理。

如果加载成功, 就调用 `calculateTotalsFor()` 方法以确保每一个任务的时间项所显示的是正确的时间。

```
StandardItem *StandardItemModel::createNewTask(QStandardItem *root,
        const QString &name, bool done)
{
    StandardItem *nameItem = new StandardItem(name, done);
    root->appendRow(QList<QStandardItem*>() << nameItem
        << nameItem->todayItem() << nameItem->totalItem());
    return nameItem;
}
```

在前面已经看到当创建一个 `StandardItem` 对象时, 在构造函数内部还创建了两个用来显示任务今天执行时间和任务所有执行时间的 `QStandardItem` 对象。一旦已经创建了新任务, 就添加一个代表这个任务 `StandardItem` 对象、连带着使用代表时间的两个 `QStandardItem` 对象, 作为指定父项之下的一个新记录行。这时, 模型取得树中所有项的所有权, 因此像 Qt 中的通常做法一样, 我们自己不用担心这些项的删除问题。

```
void StandardTreeModel::calculateTotalsFor(QStandardItem *root)
{
    if (root != invisibleRootItem()) {
        StandardItem *item = static_cast<StandardItem*>(root);
        item->todayItem()->setText(item->todayTime());
        item->totalItem()->setText(item->totalTime());
    }
    for (int row = 0; row < root->rowCount(); ++row)
        calculateTotalsFor(root->child(row, 0));
}
```

上面这个递归方法用于设置所有时间项的文本内容。这里没有介绍 `todayTime()` 或 `totalTime()` 方法, 因为它们与模型/视图编程是无关的, 另外, 在源代码中能找到这两个函数。

```
enum Insert {AtTopLevel, AsSibling, AsChild};

StandardItem *StandardItemModel::insertNewTask(Insert insert,
        const QString &name, const QModelIndex &index)
{
    QStandardItem *parent;
    if (insert == AtTopLevel)
        parent = invisibleRootItem();
    else {
        if (index.isValid()) {
            parent = itemFromIndex(index);
            if (!parent)
                return 0;
            if (insert == AsSibling)
                parent = parent->parent() ? parent->parent()
                    : invisibleRootItem();
        }
        else
            return 0;
    }
    return createNewTask(parent, name, false);
}
```



`MainWindow::editAdd()` 方法会调用上面这个方法添加新任务。首先是使用了一个枚举值 `Insert` 来确定新任务的父项到底应该是什么,然后使用在 `load()` 方法中使用过的同样的 `createNewTask()` 方法来创建一个给定了任务名(在 `editAdd()` 方法中使用的是 `New Task`)的新任务,并且设置该任务的完成状态标志为 `false`(复选框未选中)。

```
QStringList StandardTreeModel::pathForIndex(const QModelIndex &index)
const
{
    QStringList path;
    if (index.isValid()) {
        QStandardItem *item = itemFromIndex(index);
        while (item) {
            path.prepend(item->text());
            item = item->parent();
        }
    }
    return path;
}
```

上面这个方法用来返回一个任务路径,如先前所示。该方法首先添加指定项的名称到 `QStringList` 类型的 `path` 变量中,然后把它的父项的名称添加到列表的第一项,然后是祖父项的任务名称,等等,一直到最顶级。这里要记住的是,在顶级项中调用 `QStandardItem::parent()` 会返回 `0`(尽管 `QStandardItemModel` 拥有这些项的所有权)。

```
QStandardItem *StandardTreeModel::itemForPath(const QStringList &path)
const
{
    return itemForPath(invisibleRootItem(), path);
}

QStandardItem *StandardTreeModel::itemForPath(QStandardItem *root,
const QStringList &path) const
{
    Q_ASSERT(root);
    if (path.isEmpty())
        return 0;
    for (int row = 0; row < root->rowCount(); ++row) {
        QStandardItem *item = root->child(row, 0);
        if (item->text() == path.at(0)) {
            if (path.count() == 1)
                return item;
            if ((item = itemForPath(item, path.mid(1))))
                return item;
        }
    }
    return 0;
}
```

这两个方法与 `pathForIndex()` 所做的事情相反——它们根据传入的任务路径来返回对应的项。公有方法把一个任务路径作为参数,然后调用将隐藏的根项和这个任务路径作为参数的私有方法。私有方法遍历指定项的所有直接子项来寻找任务名称与任务路径列表中第一项字符串相同的那一项。如果找到了,那么该方法进行递归调用,把找到的项作为新的 `root` 参数,并且把任务路径中去除第一项(第一项已经找到匹配)后的列表作为新的任务路径。最后要么任务路径中的所有字符串都找到匹配项,返回对应的项;要么匹配失败,则返回 `0`。

现在我们已经介绍完了 `timelog1` 示例和它使用的 `QStandardItem`、`QStandardItemModel` 派生类。在下一节将创建一个自定义树模型(附加了一些功能)来进行模型替换。

4.2 创建自定义树模型

正如讨论在表格中使用 `QStandardItemModel` 时提醒注意的,使用 `QStandardItemModel` 通常是最简单和最快捷的让程序运行起来的方法。

然而,在写树模型相关的案例的时候,使用自定义树模型会比使用 `QStandardItemModel` 可能实现的功能要多。尽管如此,从 `QStandardItemModel` 着手通常是一个好方案——我们可能并不需要自定义模型提供的那些额外的功能,而且树模型比列表和表格模型复杂得多,因此使用 `QStandardItemModel` 能节省大量的工作。但是如果我们想允许用户在树中随意的移动各项——通常仅对那些具有相同类型的项并且项之间可随意嵌套(就像我们的那些任务项)的树才有意义——在这种情况下我们别无选择,只能使用自定义树模型。

在这一节中将创建一个 `timelog2` 应用程序示例,如图 4.3 所示。在这个示例中遵循前一章中介绍 `Zipcodes` 示例时的相似模式,去除一些文件并添加一些文件,使用 `#ifdef` 宏以确保能共享尽可能多的代码。在这个示例中将使用 `taskitem. {hpp,cpp}` 来代替 `standarditem. {hpp,cpp}`,并在 `timelog2.pro` 文件中加一行 `DEFINES += CUSTOM_MODEL`。

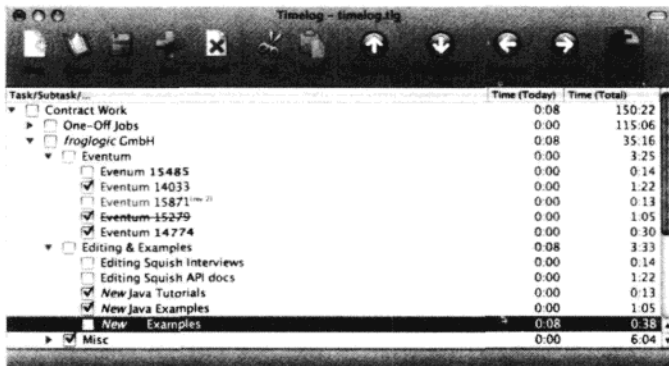


图 4.3 timelog2 应用程序示例

正如我们在 `Zipcodes` 示例中所做的,这里并不显示那些 `#ifdef` 宏,而只显示那些仅在定义了宏 `CUSTOM_MODEL` 的情况下对 `timelog2` 示例有效的代码(在前一节中,我们只介绍了在没有定义 `CUSTOM_MODEL` 宏的情况下有效的代码)。

在 4.2.1 节中先看一下在 `timelog1` 和 `timelog2` 示例之间的那些与模型/视图相关的不同之处——大多数是那些在用户界面中通过使用剪切、粘贴或者拖放操作或使用任一移动动作在树中进行项移动(包括所有直接和间接子项)有关的内容。在 4.2.2 节中,将介绍表示任务的 `TaskItem` 类,在 4.2.3 节中将介绍保存应用程序的任务数据的 `QAbstractItemModel` 的派生类 `TreeModel`。

4.2.1 通过用户界面改变树模型

在 `timelog1` 与 `timelog2` 示例的主窗口的数据成员唯一的不同之处在于, `timelog2` 使用了一个自定义的 `TreeModel`,而 `timelog1` 使用的是 `StandardTreeModel`。在成员函数方面, `timelog2` 使用了一个不同的 `hideOrShowDoneTask()` 方法,并提供了另外 6 个支持移动项功能的方法: `editCut()`、`editPaste()`、`editMoveUp()`、`editMoveDown()`、`editPromote()` 和 `editDemote()`。它们也是在用户界面中的那些相关动作触发后的响应函数。

timelog2 的构造函数仅提供了一个不同的标题,但 createModelAndView() 与前面介绍的有几处不同,下面是代码。

```
void MainWindow::createModelAndView()
{
    model = new TreeModel(this);
    treeView->setDragDropMode(QAbstractItemView::InternalMove);
    treeView->setAllColumnsShowFocus(true);
    treeView->setItemDelegateForColumn(0, new RichTextDelegate);
    treeView->setModel(model);
    setCentralWidget(treeView);
}
```

最大的变化是使用了 TreeModel。另外注意我们设置 QTreeView 支持拖放——但仅为在树内移动各项(与 timelog1 示例相同,这里没有显示在 ModelTest 模块可用的情况下那部分#ifdef 括起来的代码块。但可以在源码中找到)。委托将在第5章进行介绍。

在构造函数中调用到的 createAction() 和 createMenusAndToolBar() 方法,与 timelog1 仅有的不同是创建并使用了 timelog2 中额外支持的动作。

至于信号-槽连接, QStandardItemModel::itemChanged() 信号与 setDirty() 槽的连接中的信号被替换为 QAbstractItemModel::dataChanged()。当然还有那些新增加的动作连接到它们的响应函数——例如 editCutAction 的 triggered() 信号连接到了 editCut() 槽。

在前面介绍过的从 MainWindow::load() 中摘取出来的那段代码,在 timelog2 中稍微简单一些:

```
model->load(filename);
if (!taskPath.isEmpty()) {
    setCurrentIndex(model->indexOfPath(taskPath));
}
```

在 timelog1 示例中我们不得不通过 StandardTreeModel::itemForPath() 方法来检索某一项,然后获取该项的模型索引以供 setCurrentIndex() 方法调用,但是这里有一个 TreeModel::indexOfPath() 方法可直接使用。

其他方法与先前看到的也有些不同:editAdd() 变简短了,hideOrShowDoneTask() 变得略有不同,这里将全部进行介绍。editDelete() 方法将不做介绍,因为唯一的不同是它能与模型索引一起工作直接获取名称和要删除的子项数量,而不需要从项中获取这些信息——真正的删除工作同之前一样要调用 removeRow() 方法去处理。我们还将介绍几个新方法,会略过那些与已介绍的方法的实现几乎相同的方法。

```
void MainWindow::editAdd()
{
    QModelIndex index = treeView->currentIndex();
    if (model->insertRow(0, index)) {
        index = model->index(0, 0, index);
        setCurrentIndex(index);
        treeView->edit(index);
        setDirty();
        updateUi();
    }
}
```

在 timelog1 示例中,必须询问用户是要把任务添加到顶层、添加到当前项之后还是添加为当前项的子项。但由于 timelog2 中移动任务非常方便,所以可以把新任务添加为当前选中项的子项,而由用户根据需要决定把新添加的项移动到别的地方。

一旦创建了一个新任务(添加为当前任务项的第一个子项——或者当树为空的时候添加为顶

级项),就可以获取新添加任务项的模型索引,然后滚动到它所在的位置。接着设置为编辑状态[用户就不需要按 F2 键(在 Mac OS X 下是 Enter 键)或者双击鼠标(进入编辑模式)],这样用户就能立即输入自己的文本来代替默认的 New Task。

把新添加的项作为当前项的子项的另一个好处是它以不可见的状态创建(除非它是一个顶级任务),因此在浏览到该任务之前,视图不会做任何 data()调用。在本示例中,这不算什么真正的问题,但一般而言,当使树发生变化时隐藏项(对于父项为折叠状态的子项),通常是最好的,因为这能避免视图在可能进行过移动或删除操作的项上调用 data()方法。

```
void MainWindow::editHideOrShowDoneTasks(bool hide)
{
    hideOrShowDoneTask(hide, QModelIndex());
}

void MainWindow::hideOrShowDoneTask(bool hide,
                                     const QModelIndex &index)
{
    bool hideThisOne = hide && model->isChecked(index);
    if (index.isValid())
        treeView->setRowHidden(index.row(), index.parent(),
                                hideThisOne);

    if (!hideThisOne) {
        for (int row = 0; row < model->rowCount(index); ++row)
            hideOrShowDoneTask(hide, model->index(row, 0, index));
    }
}
```

这两个方法与 `timelog1` 中的同名方法十分相似并且使用了相同的逻辑。最关键的不同在于依照模型的索引而非项来进行处理,因为使用了 `QAbstractItemModel` API(加上我们自己的逻辑扩展,如 `isChecked()` 方法),而不是基于项的 `QStandardItemModel` API。

```
void MainWindow::editCut()
{
    QModelIndex index = treeView->currentIndex();
    if (model->isTimedItem(index))
        stopTiming();
    setCurrentIndex(model->cut(index));
    editPasteAction->setEnabled(model->hasCutItem());
}
```

这个槽方法像所有与移动相关的槽一样,把大部分工作交给了模型去处理。这是必要的,因为当删除或移动任务时,树的结构发生了变化,而这必须要在所有相关联的视图中体现出来。

如果要剪切的任务是正在计时(即正在执行的)的,就要先停止计时,因为对于隐藏的并且可能被删除(如果没有被粘贴回去的话)的任务进行计时没有意义。然后通知模型把指定模型索引对应的任务项剪切掉(递归地告知它的所有子孙项)并设置返回的模型索引代表的任务为选中项。还要更新粘贴动作的可用状态以使用户知道当前粘贴操作可用。

```
void MainWindow::editPaste()
{
    setCurrentIndex(model->paste(treeView->currentIndex()));
    editHideOrShowDoneTasks(
        editHideOrShowDoneTasksAction->isChecked());
}
```

这个槽是 `timelog2` 示例提供的几乎所有的移动槽的典型代表:它告诉模型执行粘贴操作,然后滚动到模型返回的模型索引代表的任务项的位置并选中它,最后隐藏或显示已完成任务,以确保显示(或隐藏)与 `editHideOrShowDoneTasksAction` 的状态保持一致。

粘贴回来的这个任务与它的所有子项(递归的)一起添加进来,但是它的子项都是折叠起来的不可见的,直到用户展开它们时为止。

`editMoveUp()`槽方法与上面介绍的几乎相同(唯一的不同之处是需要调用 `TreeModel::moveUp()`来代替 `TreeModel::paste()`)。对于 `editMoveDown()`也是一样的,调用了 `TreeModel::moveDown()`方法。`editPromote()`和 `editDemote()`方法(调用了 `TreeModel::promote()`和 `TreeModel::demote()`)除了如果当前项正在计时的话要先停止计时(正如我们 `editCut()`中所做的)外也非常相似。在所有情况下,这些操作都递归的作用于当前选中任务及它的所有子项。

现在已经介绍了足够的与用户界面相关的方法和槽,以给我们需要的上下文信息来理解用来保存任务数据的 `TreeModel`。但是在看 `TreeModel` 的实现之前,必须先看一下 `TreeModel` 内部用来代表任务的 `TaskItem` 类。

4.2.2 用于树中各项的自定义类

我们需要一个类来代表树中的各项,当使用 `QStandardItem` 的时候,需要为任务名称、当日任务执行时间和总体任务执行时间分别创建项。但由于将使用一个自定义模型,所以能够把需要的所有数据保存在一个单独的 `TaskItem` 中,让模型在需要的时候返回每一列对应的数据。

`TaskItem` 类里提供了两类函数:处理数据(任务名称、完成状态和起始-结束时间段)的方法和管理子项的方法。如在下一小节中将要看到的,整个树实际上是通过一个指向根项(`root item`,在 `TreeModel` 中是一个未命名的任务项,等同于 `QStandardItemModel` 中的隐藏的根项)的指针来代表的,其他所有的任务都是这个根项的直接或间接子项。

大多数 `TaskItem` 类的方法都在头文件中定义了,我们把头文件分成三部分来进行介绍(两组方法,最后是私有数据成员),先从那些组成方法基础的私有数据成员开始介绍。

```
private:
    int minutesForTask(bool onlyForToday) const;
    QString m_name;
    bool m_done;
    QList<QPair<QDateTime, QDateTime> > m_dateTimes;
    TaskItem *m_parent;
    QList<TaskItem*> m_children;
};
```

这里没有介绍计算任务执行时间的方法,因为它与模型/视图编程无关。每一个任务都有反映存储在 XML 文件中的数据(任务名称、完成状态、起始-结束时间段)的数据成员。另外,为支持树的层次结构,每一个任务项还有一个指向父项的指针和一个直接子项的列表。未命名的根项(从来不会加载、保存,只是为编程方便而存在)是仅有的一个父项为 0 的项。

对比 `TaskItem` 中保存在 `TaskItem` 的数据, `timelog1` 的 `QStandardItemModel` 版本里 `StandardItem` 则需要额外的两个 `QStandardItem` 项。然而在两个示例中,自定义数据要保存的内容是相同的(任务名称、完成状态、起始-结束时间段列表)。就系统开销来说,一个 `TaskItem` 对象加了一个指针和一个保存指针的 `QList` 列表对象成员,然而 `QStandardItemModel` 需要三个 `QStandardItem`,这样的话就添加了整整 9 个指针、9 个整型、3 个保存了指针的 `QVector`,3 个保存了值(每一个值都包含有一个整数和一个 `QVariant` 对象)的 `QVector`(这是基于 Qt 4.5.0 的数据)。当然了,是否会在意这些多出来的内存问题,则依赖于应用程序本身。即使如此(系统开销较大),从 `QStandardItemModel` 开始仍然几乎总是更好的选择,并且只有在性能或功能方面有必要的时候才实现自定义的 `QAbstractItemModel`。

```
class TaskItem
{
public:
    explicit TaskItem(const QString &name=QString(), bool done=false,
                     TaskItem *parent=0);
```

```

~TaskItem() { qDeleteAll(m_children); }

QString name() const { return m_name; }
void setName(const QString &name) { m_name = name; }
bool isDone() const { return m_done; }
void setDone(bool done) { m_done = done; }
QList<QPair<QDateTime, QDateTime> > dateTimes() const
{ return m_dateTimes; }
void addDateTime(const QDateTime &start, const QDateTime &end)
{ m_dateTimes << qMakePair(start, end); }
QString todaysTime() const;
QString totalTime() const;
void incrementLastEndTime(int msec);

```

一会将看到构造函数;但我们不介绍计算每一项的当天和总共任务执行时间以及累加最后结束时间的方法。

我们必须实现一个析构函数,因为 TaskItem 不是一个 QObject 子类,所以它需要自己掌控任务项的所有权。当一个项被删除时要删除它的所有直接子项,然后这些子项会依次递归地删除它们各自的直接子项,因此只需要删除树的根项就可以删除掉所有的任务项了。

```

TaskItem *parent() const { return m_parent; }
TaskItem *childAt(int row) const { return m_children.value(row); }
int rowOfChild(TaskItem *child) const
{ return m_children.indexOf(child); }
int childCount() const { return m_children.count(); }
bool hasChildren() const { return !m_children.isEmpty(); }
QList<TaskItem*> children() const { return m_children; }

void insertChild(int row, TaskItem *item)
{ item->m_parent = this; m_children.insert(row, item); }
void addChild(TaskItem *item)
{ item->m_parent = this; m_children << item; }
void swapChildren(int oldRow, int newRow)
{ m_children.swap(oldRow, newRow); }
TaskItem* takeChild(int row);

```

childAt() 方法小心地使用了 QList::value() 方法而不是 QList::operator[](), 这是为了确保如果给出了一个超出范围的行参数,就构造一个默认值(即 0)返回,否则程序会崩溃。

当一个项作为子项插入到特定行或添加到它的尾部,就有必要指定该子项的父项为当前项。这是因为 TaskItem 可能已被从树中某处剪切或移动,但它以前的父项如果与现在的父项不同(通常情况下不同),那么就会出错。

在 taskitem.cpp 文件中只有两个方法是与模型/视图编程相关的:构造函数和 takeChild() 方法,下面都将进行介绍。

```

TaskItem::TaskItem(const QString &name, bool done, TaskItem *parent)
: m_name(name), m_done(done), m_parent(parent)
{
    if (m_parent)
        m_parent->addChild(this);
}

```

如果新创建的任务项具有非空父项,那么就将它加入到其父项中子项列表的末尾。

```

TaskItem* TaskItem::takeChild(int row)
{
    TaskItem *item = m_children.takeAt(row);
    Q_ASSERT(item);
    item->m_parent = 0;
    return item;
}

```

如果一个任务项已被从树中拿去,即从它的父项的子项列表中移除,必须设置这个任务项的父项

为 0 以表示它不再属于任何项的事实。这意味着返回的指针是我们的责任,我们必须尽快删除它或者把它插入到树中,以防止内存泄漏。

4.2.3 用于树的自定义 QAbstractItemModel 派生类

为树实现一个既能编辑又能添加/删除行的 QAbstractItemModel 派生类,我们通常必须要实现表 3.1 列出来的所有或几乎所有的方法。然而,对于 Timelog 应用程序的任务数据(通常针对树来说)不需要重新实现 insertColumns() 或 removeColumns(),因为用到的列数是固定的。

为了支持使用拖放来移动项(包括所有子项)我们也要实现表 4.1 中列出来的拖放相关的方法。拖放 API 需要我们对模型项进行序列化(serialize)和反序列化(deserialize),正如我们将要看到的,可以用加载和保存数据时使用的方法来帮助实现。我们还希望提供给用户其他的移动项的方法,为此用一些自定义方法扩展了 QAbstractItemModel 的 API。

```
QModelIndex moveUp(const QModelIndex &index);
QModelIndex moveDown(const QModelIndex &index);
QModelIndex cut(const QModelIndex &index);
QModelIndex paste(const QModelIndex &index);
QModelIndex promote(const QModelIndex &index);
QModelIndex demote(const QModelIndex &index);
```

表 4.1 QAbstractItemModel 的拖放 API

方 法	说 明
dropMimeData(mimeData, dropAction, row, column, parent)	当放下动作发生时调用该方法;它必须反序列化 mimeData 并使用它在指定项(行为 row、列为 column、父项为 parent)上执行 dropAction 动作
mimeData(indexes)	返回一个 QMimeData 对象,该对象包含了参数 indexes(模型索引列表)对应的序列化的数据;它在模型内部被用来产生放下动作所需要的数据
mimeTypes()	返回 QStringList 形式的 MIME 类型列表用于描述模型索引
supportedDragActions()	返回一个或多个 Qt::DropAction 类型的值的按位或(没有拖曳操作的枚举值)
supportedDropActions()	返回一个或多个 Qt::DropAction 类型的枚举值的按位或(Qt::CopyAction、Qt::MoveAction 等)

Qt 4.6 引入了 4 个新的保护方法以简化在模型中移动项的操作:beginMoveColumns()、endMoveColumns()、beginMoveRows() 和 endMoveRows()。为了保持代码在 Qt 4.5 与 Qt 4.6 之间的兼容,我们没有使用这几个函数。对于要求 Qt 最小版本 4.6 的项目,这些新的保护方法有可能有用,但要仔细阅读它们的文档,因为有一些限制条件。

除了附加的自定义方法外,TreeModel 类还有与计时项(timing item)相关的一些方法和数据(这里未做介绍),及与添加到 QStandardItemModel 中的那些相同的方法(或等价方法):clear()、load()、save()、pathForIndex() 和 indexForPath()。

TreeModel 有各种各样的私有方法,当在讨论公有方法和一些私有数据的时候,如果有需要,我们将进行介绍。

```
private:
    QString m_filename;
    QIcon m_icon;
    TaskItem *timedItem;
    TaskItem *rootItem;
    TaskItem *cutItem;
```

对于那些 TaskItem 项,timedItem 是一个有父项的指针,所以不必担心它的删除问题。rootItem 代表

树的根项,我们必须在适当的时候删除它。cutItem 表示一个已被剪切但还未被粘贴的项,如果这样的项存在,就必须在合适的时候删除它——例如当打开一个新文件,或者应用程序被终止。

所有的这些都意味着 TreeModel 能够取代 StandardTreeModel,但要加上那些附加的功能,尤其是支持剪切/粘贴、拖放和移动项的功能。

我们将介绍 TreeModel 的所有模型/视图相关的方法。先从构造函数和析构函数开始,然后看一下实现 QAbstractItemModel API 的方法,接着是 isChecked() 方法(这是为了完整性,因为我们在前面使用过它),实现 QAbstractItemModel 的拖放 API 的函数、实现移动的函数,最后是文件和任务路径的处理方法。

```
explicit TreeModel(QObject *parent=0)
    : QAbstractItemModel(parent), timedItem(0), rootItem(0),
      cutItem(0) {}

~TreeModel() { delete rootItem; delete cutItem; }
```

构造函数仅仅需要初始化数据成员指针为 0,并把 parent 参数传给基类。析构函数就必须删除根项和剪切项(cut item,它可能是 0)。我们依赖 TaskItem 的析构函数来递归地删除每一个任务项的所有子项(直接和间接的)。

4.2.3.1 用于树的 QAbstractItemModel API

在这一小节中将介绍 TreeModel 实现的、QAbstractItemModel API 中用于为树提供编辑和插入/删除项(按照行,而非列)的方法。这些方法在表 3.1 中列了出来。

```
enum Column {Name, Today, Total};

Qt::ItemFlags TreeModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags theFlags = QAbstractItemModel::flags(index);
    if (index.isValid()) {
        theFlags |= Qt::ItemIsSelectable|Qt::ItemIsEnabled;
        if (index.column() == Name)
            theFlags |= Qt::ItemIsUserCheckable|Qt::ItemIsEditable|
                Qt::ItemIsDragEnabled|Qt::ItemIsDropEnabled;
    }
    return theFlags;
}
```

这个方法的实现对所有类型的模型都是相似的。在这个例子中设置所有的项为可选择的(selectable)和可用的(enabled),但只允许代表任务名称的项是可复选的(checkable)、可编辑的(editable)和可拖放的(dragged and dropped)。就如即将看到的,我们实现了拖放功能,如任务名称被拖动,整个任务包括时间项也将被拖动,而且所有的子项(直接和间接的)也将被拖动。但我们更希望用户拖动任务名称项而不是时间,因为这样能清晰地表明正在发生的操作。

```
const int ColumnCount = 3;

QVariant TreeModel::data(const QModelIndex &index, int role) const
{
    if (!rootItem || !index.isValid() || index.column() < 0 ||
        index.column() >= ColumnCount)
        return QVariant();
    if (TaskItem *item = itemForIndex(index)) {
        if (role == Qt::DisplayRole || role == Qt::EditRole) {
            switch (index.column()) {
                case Name: return item->name();
                case Today: return item->todayTime();
                case Total: return item->totalTime();
            }
        }
    }
}
```



```

        default: Q_ASSERT(false);
    }
}
if (role == Qt::CheckStateRole && index.column() == Name)
    return static_cast<int>(item->isDone() ? Qt::Checked
                               : Qt::Unchecked);
if (role == Qt::TextAlignmentRole) {
    if (index.column() == Name)
        return static_cast<int>(Qt::AlignVCenter|
                                Qt::AlignLeft);
    return static_cast<int>(Qt::AlignVCenter|Qt::AlignRight);
}
if (role == Qt::DecorationRole && index.column() == Today &&
    timedItem && item == timedItem && !m_icon.isNull())
    return m_icon;
}
return QVariant();
}

```

`data()`方法是Qt的模型/视图架构中的关键,因为通过这个方法可以访问所有的数据和大部分元数据。如在讨论TableModel时提到的,这个方法不依赖于对于基类实现的调用,而是要求我们必须对自己不处理的情况总是返回一个无效的QVariant。

`itemForIndex()`方法根据参数指定的模型索引返回树中对应项的TaskItem指针,过一会来介绍这个方法。

我们选择对Qt::DisplayRole和Qt::EditRole同样对待,因此当数据的角色为其中之一时,直接返回对应的数据。这里的数据没有列的概念,而只有一个由任务项组成的树,但把列映射到数据的特定字段,或者在本例为时间列中显示的计算值。我们还处理了Qt::CheckStateRole角色,返回对应任务完成状态的枚举值。

对于文本对齐方式的角色,这里选择把任务名称项左对齐,时间项右对齐。还为请求的Qt::DecorationRole返回了一个图标——但只对请求的是“Today”这列的数据、且该项正在计时的情况。在其他情况下以及对于其他角色,就直接返回一个无效的QVariant,即交给Qt来处理。

```

TaskItem *TreeModel::itemForIndex(const QModelIndex &index) const
{
    if (index.isValid()) {
        if (TaskItem *item = static_cast<TaskItem*>(
            index.internalPointer()))
            return item;
    }
    return rootItem;
}

```

无论何时创建一个QModelIndex(使用QAbstractItemModel::createIndex()方法),除了要提供行号和列号,还可以提供一个指针(或者一个数字ID)。针对树模型,普遍的情况是提供一个指向树中对应项的指针——就如将要看到的,我们在TreeModel中创建模型索引时也是这么做的。这使得按给定的模型索引获取一个指向对应项的指针轻而易举——我们只要获取模型索引的内部指针就可以了。如果没有指针,或者索引是无效的,就会返回一个指向根项的指针(如果没有任何项添加到树中的话,这个指针的值就是0)。

```

QVariant TreeModel::headerData(int section,
    Qt::Orientation orientation, int role) const
{
    if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {
        if (section == Name)
            return tr("Task/Subtask/...");
        else if (section == Today)
            return tr("Time (Today)");
    }
}

```

```

        else if (section == Total)
            return tr("Time (Total)");
    }
    return QVariant();
}

```

Qt 中的 `QTreeView` 部件只支持横表头 (horizontal header), 所以当请求横表头的数据的时候要为其提供一个合适的名称。由于 `headerData()` 使用了与 `data()` 函数相同的逻辑, 也就是说, 依靠返回值而不调用基类的方法, 对于那些没有处理的情形返回一个无效的 `QVariant` 对象。

```

int TreeModel::rowCount(const QModelIndex &parent) const
{
    if (parent.isValid() && parent.column() != 0)
        return 0;
    TaskItem *parentItem = itemForIndex(parent);
    return parentItem ? parentItem->childCount() : 0;
}

```

树中的某项的行数是指它的所有直接子项的数目 (不包含间接子项, 所以这个数目并不是递归的)。如果父项是合法的但列号不是 0, 就返回 0, 因为我们只允许第一列的项能够有子项。否则获取参数 `parent` 索引所对应的任务项。如果 `parent` 索引是无效的, 那么 `itemForIndex()` 会正确地返回根项 (如果树中没有任何项, 就会返回 0)。

```

int TreeModel::columnCount(const QModelIndex &parent) const
{
    return parent.isValid() && parent.column() != 0 ? 0 : ColumnCount;
}

```

自定义的 `TreeModel`, 与许多其他树模型一样, 有固定的列数, 所以实现上面这个方法就比较容易, 与我们在自定义 `Zipcodes` 表格模型实现的方法相似。如果给定的索引是有效的并且不是第一列 (名称列), 就请求获取时间列的列数, 但这对于这个模型没有意义, 所以这种情况下直接返回 0。

```

QModelIndex TreeModel::index(int row, int column,
                              const QModelIndex &parent) const
{
    if (!rootItem || row < 0 || column < 0 || column >= ColumnCount
        || (parent.isValid() && parent.column() != 0))
        return QModelIndex();
    TaskItem *parentItem = itemForIndex(parent);
    Q_ASSERT(parentItem);
    if (TaskItem *item = parentItem->childAt(row))
        return createIndex(row, column, item);
    return QModelIndex();
}

```

这个方法用来给模型使用者提供模型索引, 同时模型内部也使用它。

除了明显的有效性测试外, 还要检查参数 `parent` 所在的列号。这里没有为 `parent` 的列的不为 0 的项提供模型索引, 因为只允许列号为 0 的项能够有子项。

一个模型索引由行号、列号和一个指针 (或一个数字 ID) 构成。对于列表和表格模型而言, 这个指针通常是 0, 但对于树模型而言, 它通常是指向树中相应项的指针 (或 ID)。在这里先使用 `itemForIndex()` 方法获取参数 `parent` 所指的的任务项, 然后获取该任务项的行号为 `row` 的直接子项。接着调用 `QAbstractItemModel::createIndex()` 来获得模型索引, 它的参数为 `row` 和 `column` 以及刚才获取的子项指针, 因为这个子项是这个新建的模型索引真正对应的项——这样这个指针就成为了这个模型索引的内部指针。

如果不能创建这个索引, 就必须返回一个无效的 `QModelIndex`。需要注意的是, `QModelIndex` 只有两个公有构造函数——一个拷贝构造函数和一个没有任何参数、只能用来创建无效模型索引

的构造函数。所以唯一的创建有效模型索引的方法是调用 `createIndex()` 方法, 或者使用拷贝构造函数来复制已存在的模型索引。

这个方法的结构——也同样是大部分这里介绍的其他实现 `QAbstractItemModel` API 的树模型方法的结构——可以应用于任何使用项的指针树, 同时该项的类型(例如 `TaskItem` 和它的方法或它的等价类型)提供了子项操作的方法的树模型, 该模型还有 `itemForIndex()` (或它的等价方法)。因此这里所介绍的代码应该能直接应用到其他需要的地方。

```
QModelIndex TreeModel::parent(const QModelIndex &index) const
{
    if (!index.isValid())
        return QModelIndex();
    if (TaskItem *childItem = itemForIndex(index)) {
        if (TaskItem *parentItem = childItem->parent()) {
            if (parentItem == rootItem)
                return QModelIndex();
            if (TaskItem *grandParentItem = parentItem->parent()) {
                int row = grandParentItem->rowOfChild(parentItem);
                return createIndex(row, 0, parentItem);
            }
        }
    }
    return QModelIndex();
}
```

尽管 `TaskItem` 有一个指向它父项的指针, 但返回一个项的父项对应的模型索引并不是看上去那么简单。这是因为无法直接从一个 `TaskItem` 指针映射到它对应的模型索引。所以应先找到该项的父项, 然后再找到这个父项在它自己的父项的直接子项列表中的位置(也就是找到该项的父项在该项的祖父项的子项列表中的行号)。一旦知道了这个行号并且有了该项的父项对应的指针, 就可以用 `createIndex()` 来创建该项的父项的模型索引了。

注意, 如果父项是根项的话, 返回一个无效的模型索引——按照惯例, Qt 的模型/视图架构使用一个无效的模型索引(而不是根项, 如果模型有根项的话)来代表顶级项的父项索引, 所以我们也确保代码要这样实现。

图 4.4 演示了项、父项以及行号之间的关系。在图中, 项 A 的父项是项 P, 这就是说, 项 A 是项 P 的第一个(即行号为 0)直接子项, 并且项 P 的父项是项 GP, 也即项 P 是项 GP 的第二个(即行号为 1)直接子项。

```
bool setHeaderData(int, Qt::Orientation, const QVariant&,
    int=Qt::EditRole) { return false; }
```

我们使表头为只读的, 上面的简单实现放在头文件中就可以了。

```
bool TreeModel::setData(const QModelIndex &index,
    const QVariant &value, int role)
{
    if (!index.isValid() || index.column() != Name)
        return false;
    if (TaskItem *item = itemForIndex(index)) {
        if (role == Qt::EditRole)
            item->setName(value.toString());
        else if (role == Qt::CheckStateRole)
            item->setDone(value.toBool());
        else
            return false;
    }
}
```

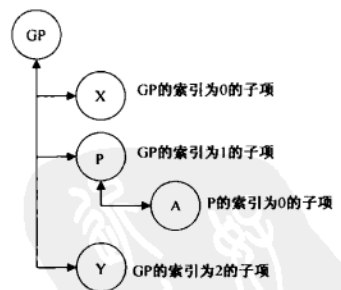


图 4.4 项的父项是它的祖父项的第 n (行号) 个子项

```

        emit dataChanged(index, index);
        return true;
    }
    return false;
}

```

我们使用上面这个方法的支持对于任务项(具体是任务名称和完成状态)的编辑。我们并不需要关心编辑的细节——自定义的富文本委托对象会处理任务名称的编辑工作,还会使用一个复选框来切换完成状态。

如果发生编辑动作,就需要发射带有数据发生变化的项的模型索引作为参数的 `dataChanged()` 信号并且返回 `true`; 否则返回 `false`。第一个模型索引是发生改变的矩形区域的左上位置对应的索引,第二个模型索引是该矩形区域的右下位置对应的索引。在这个示例中同一时间只能编辑一个模型索引,所以这两个参数使用同一个模型索引。

```

bool TreeModel::insertRows(int row, int count,
                           const QModelIndex &parent)
{
    if (!rootItem)
        rootItem = new TaskItem;
    TaskItem *parentItem = parent.isValid() ? itemForIndex(parent)
                                           : rootItem;
    beginInsertRows(parent, row, row + count - 1);
    for (int i = 0; i < count; ++i) {
        TaskItem *item = new TaskItem(tr("New Task"), false);
        parentItem->insertChild(row, item);
    }
    endInsertRows();
    return true;
}

```

Qt 的模型/视图 API 需要在向模型中插入任何行之前调用 `QAbstractItemModel::beginInsertRows()`, 在插入行操作完成后调用 `QAbstractItemModel::endInsertRows()`。 `beginInsertRows()` 调用可以原封不动地复制到其他实现中——它的参数为父项模型索引、新行插入位置的行号和最后一个新行的行号。

首先要确保有一个根项——举例来说,如果用户刚刚选择了 `File→New` 菜单,则根项还未被创建。然后获取将要插入的所有项的父项——可能是给定的 `parent` 参数所代指的项(如果这个 `parent` 是有效的),也可能是根项(这样,所有要插入的项就是顶级项)。之后创建 `count` 个新任务项,每个任务项都用默认的名称和默认完成状态,并且每个都是在 `parent` 所代指的父项的一个指定行位置插入,从而成为该父项下的一个直接子项(回想一下 `TaskItem` 的构造函数,如果父项索引参数不为 `null` 的话,子项就会把自身插入到父项的直接子项列表中)。

当一个任务项(包括它的所有直接和间接子项)被拖放到新位置时,这个方法在幕后被调用。要记得还有个 `insertRow()` 方法(在 `MainWindow::editAdd()` 方法中用到了),但我们不需要重新实现它,因为基类的实现已正确地调用插入行数为 1 作为参数的 `insertRows()` 方法。

```

bool TreeModel::removeRows(int row, int count,
                           const QModelIndex &parent)
{
    if (!rootItem)
        return false;
    TaskItem *item = parent.isValid() ? itemForIndex(parent)
                                       : rootItem;
    beginRemoveRows(parent, row, row + count - 1);
    for (int i = 0; i < count; ++i)
        delete item->takeChild(row);
    endRemoveRows();
    return true;
}

```

Qt 的模型/视图 API 需要在我们从模型中删除任何行之前先调用 `QAbstractItemModel::beginRemoveRows()` 方法,并在删除操作完成后调用 `QAbstractItemModel::endRemoveRows()` 方法。`beginRemoveRows()` 的调用可以原封不动地应用到其他应用程序的实现中。

如果没有根项,那么树就是空的,所以没有什么可删除的东西——既然如此那就什么也不用做直接返回 `false`。否则要从参数 `row` 所指的行开始,删除 `count` 行,我们简单地删除指定行的项 `count` 次。第一次删除操作删除了第 `row` 行的项,第二次删除操作删除了第 `row + 1` 行的项(由于前一次删除操作,该项已经成为目前的第 `row` 项),依次类推。自然地,被删除项的直接子项、间接子项,等等,都会被删除掉。需要注意的是, `TaskItem::takeChild()` 方法把指定的项从它的父项处移除,设置它的父项索引属性为 0,并把它返回(此时这个项既没有父项也没有所有者)——我们立刻把它销毁(析构)。

当某一个任务项(包括它的直接和间接子项)在拖放结束后,原先拖动的那些行就会通过调用这个方法在幕后删除掉——放下后的新行是使用 `insertRows()` 方法新添加的。

现在,我们已经介绍完了所有提供一个可编辑的、可插入/删除项(在这里是插入、删除行)的树模型所必须重新实现的方法。

下面来看一个用来扩展 `QAbstractItemModel` API 的小方法,因为这个方法在前面见过,然后介绍一组支持拖放、移动操作的辅助方法,以及从磁盘中加载和保存数据到磁盘的方法。

```
bool TreeModel::isChecked(const QModelIndex &index) const
{
    if (!index.isValid())
        return false;
    return data(index, Qt::CheckStateRole).toInt() == Qt::Checked;
}
```

这是一个我们添加的一个简便方法,为了使 `MainWindow::hideOrShowDoneTask()` 方法的实现更加容易被读懂。除了必须实现相关的 `QAbstractItemModel` API,也可以自由地提供一些像 `isChecked()` 之类的额外的便利方法。

4.2.3.2 用于拖放的 `QAbstractItemModel` API

在这一小节将介绍 `TreeModel` 实现的提供拖放功能的 `QAbstractItemModel` API 方法。这些方法在表 4.1 中列出。Qt 的拖放操作的原理是,在拖动项时必须进行数据的序列化,在放下时就要进行对应的数据反序列化^①。

```
Qt::DropActions supportedDragActions() const
{ return Qt::MoveAction; }
Qt::DropActions supportedDropActions() const
{ return Qt::MoveAction; }
```

这两个方法的实现都放在了头文件中。这里必须特别指出的是,程序只支持拖放动作中的移动功能。这对于任务数据是非常合适的,但对于其他类型的数据,就需要着重考虑提供复制功能,或者同时支持移动和复制功能(可以通过返回 `Qt::MoveAction | Qt::CopyAction` 来达到)。

```
const QString MimeType = "application/vnd.qtrac.xml.task.z";
QStringList TreeModel::mimeTypes() const
{
    return QStringList() << MimeType;
}
```

Qt 的拖放系统[也包括剪贴板(clipboard)处理]使用 MIME 类型来标志数据(在前面的章节中简单

① 需要注意的是,这段代码能在 Qt 4.5 及以后的版本中运行,但应该不能在 Qt 4.4 及以前的版本中运行。

地讨论过 MIME 类型)。必须重新实现 `mimeType()` 方法来返回自定义模型支持的 MIME 类型。这里已经创建了一个自定义的 MIME 类型来标志任务数据。就如过一会就会看到的,使用与保存和加载任务数据时相同的 XML 格式来支持拖放任务数据。这不像使用紧凑的二进制格式那样有效率,但是有一个优点就是可以使用在加载和保存任务数据时使用的序列化和反序列化部分一样的代码。

```
const int MaxCompression = 9;

QMimeType *TreeModel::mimeType(const QModelIndexList &indexes) const
{
    Q_ASSERT(indexes.count());
    if (indexes.count() != 1)
        return 0;
    if (TaskItem *item = itemForIndex(indexes.at(0))) {
        QMimeType *mimeType = new QMimeType;
        QByteArray xmlData;
        QXmlStreamWriter writer(&xmlData);
        writeTaskAndChildren(&writer, item);
        mimeType->setData(MimeType, qCompress(xmlData,
                                              MaxCompression));
        return mimeType;
    }
    return 0;
}
```

这个方法是在拖动发起时自动调用的,传入了用户开始拖动的项的模型索引列表作为参数。在树模型中,如果拖动一项,那么该项的索引将被放入到这个模型索引列表中(参数 `indexes`)——但是它的直接或间接子项的索引不会在索引列表中,尽管它们(所有直接和间接的子项)也会随它的父项一起被拖动。

`mimeType()` 方法支持在拖放多个项的情况下使用(因为它的参数是一个模型索引列表)。然而我们只处理一个项的拖动(尽管这还包含它的所有直接和间接子项)。先获取这个项的指针,然后创建一个 `QByteArray` 类型的变量用来写入 XML 格式(与用来保存在磁盘中的任务数据的 XML 文件相同的格式)的任务数据。然后使用 `QXmlStreamWriter` 类把任务及其所有子项的数据保存为 XML 格式数据(图 4.2 显示了这些 XML 数据的格式)。当数据保存完毕后,把数据用最大压缩率进行压缩(即速度最慢,数据最紧凑)——以此来减少内存消耗,因为我们使用了冗长的 XML 格式——并把结果数据作为 `QMimeType` 对象的数据。最后这个方法返回这个 `QMimeType` 对象,剩下的交由 Qt 去处理,这样我们就不必担心它的删除问题了。

```
void TreeModel::writeTaskAndChildren(QXmlStreamWriter *writer,
                                     TaskItem *task) const
{
    if (task != rootItem) {
        writer->writeStartElement(TaskTag);
        writer->writeAttribute(NameAttribute, task->name());
        writer->writeAttribute(DoneAttribute, task->isDone() ? "1"
                               : "0");

        QListIterator<
            QPair<QDateTime, QDateTime> > i(task->dateTimes());
        while (i.hasNext()) {
            const QPair<QDateTime, QDateTime> &dateTime = i.next();
            writer->writeStartElement(WhenTag);
            writer->writeAttribute(StartAttribute,
                                   dateTime.first.toString(Qt::ISODate));
            writer->writeAttribute(EndAttribute,
                                   dateTime.second.toString(Qt::ISODate));
            writer->writeEndElement(); // WHEN
        }
    }
}
```



```

    }
}
foreach (TaskItem *child, task->children())
    writeTaskAndChildren(writer, child);
if (task != rootItem)
    writer->writeEndElement(); // TASK
}

```

这个方法用来把一项的数据以 XML 格式保存到给定的 QXmlStreamWriter 对象中。它和我们在 StandardTreeModel 中创建的同名函数几乎是一模一样的。

我们从来不会输出没有命名的根项,因为它只是为编程方便而存在的,而不是数据的一部分。当指定项的数据输出完毕后,把所有项的直接和间接子项递归到输出,所以如果指定根项为参数的话,这个方法可以用来输出整个树——尽管这里它只用来输出一个拖动的项(及其所有子项)。

```

bool TreeModel::dropMimeData(const QMimeData *mimedata,
    Qt::DropAction action, int row, int column,
    const QModelIndex &parent)
{
    if (action == Qt::IgnoreAction)
        return true;
    if (action != Qt::MoveAction || column > 0 ||
        !mimedata || !mimedata->hasFormat(MimeType))
        return false;
    if (TaskItem *item = itemForIndex(parent)) {
        emit stopTiming();
        QByteArray xmlData = qUncompress(mimedata->data(MimeType));
        QXmlStreamReader reader(xmlData);
        if (row == -1)
            row = parent.isValid() ? parent.row()
                                   : rootItem->childCount();
        beginInsertRows(parent, row, row);
        readTasks(&reader, item);
        endInsertRows();
        return true;
    }
    return false;
}

```

这个方法是在放下动作发生时自动调用的。如果参数 action 所代指的动作是可接受的(在本例中是移动),那么首先获取数据被放下的位置的项。拖放的工作方式是删除被拖动的那些项,在放下时再创建与拖动的项相匹配的项。这意味着在拖放结束后那些指向被拖动的项的指针都不再有效。为了处理这种情况,程序发射了一个自定义信号 stopTiming(),告诉任何与它连接的 QObject 对象(在这里是 MainWindow)在放下动作发生时,正在计时的那些项停止更新时间(当然,也可以先检查是否有正在计时的项在拖动的项中,如果有的话才发射出 stopTiming()信号,但我们的方法更快并且提供了始终如一的行为)。

一旦有了要放下的项的父项,QMimeData 中的数据就解压成 XML 格式。然后调用 readTasks() 方法来重新创建放下的任务项(包括所有直接和间接子项)并把它作为那个被放下的父项的子项。

如果发生了放下动作,就必须返回 true,否则返回 false。对于移动项的情况,Qt 在背后会调用 removeRows() 方法来删除原先被拖动的那些项。

我们并不知道到底有多少行会插入到模型中,因为不知道 XML 数据描述的项是否带有子项。这没有关系,因为从视图的角度看,放下操作时只添加一个可见或不可见(取决于放下的位置是一个折叠的还是展开的项)的子项,而不管这个项到底有多少子项,因为任何子项都是折叠状态。如果给定的 row 参数是有效的,就用它作为插入行,否则插入到父项所在位置,或者父项索引无效,

就作为最后一个顶级项插入。调用 `beginInsertRows()` 和 `endInsertRows()` 对于防止视图混乱是非常重要的。

这里注意到对于检查那些拖放过来要插入的模型的行来说,模型测试显得过多了,我们需要把两行注释掉以避免错误警告的断言。

遗憾的是,一些平台下在树视图中进行拖放操作可能会不太稳定。例如,当在 Linux 下使用 Qt 4.5 在树中进行拖放操作时,不需要太费劲就可能导致崩溃。在 Mac OS X 平台(在 Qt 4.5 和 Qt 4.6 下都是)下,尽管在大多数情况下拖放操作都工作得很好,但有时候不能放下到某几个开头的项中。幸运的是,这些问题都没有影响到 Qt 在 Windows 平台下的表现,在任何情况下,使用工具栏按钮或者键盘按键来进行项的移动和提升、降级操作与用拖放操作来移动项的方式提供给用户同样多的自由度。

```
void TreeModel::readTasks(QXmlStreamReader *reader, TaskItem *task)
{
    while (!reader->atEnd()) {
        reader->readNext();
        if (reader->isStartElement()) {
            if (reader->name() == TaskTag) {
                const QString name = reader->attributes()
                    .value(NameAttribute).toString();
                bool done = reader->attributes().value(DoneAttribute)
                    == "1";
                task = new TaskItem(name, done, task);
            }
            else if (reader->name() == WhenTag) {
                const QDateTime start = QDateTime::fromString(
                    reader->attributes().value(StartAttribute)
                        .toString(), Qt::ISODate);
                const QDateTime end = QDateTime::fromString(
                    reader->attributes().value(EndAttribute)
                        .toString(), Qt::ISODate);
                Q_ASSERT(task);
                task->addDateTime(start, end);
            }
        }
        else if (reader->isEndElement()) {
            if (reader->name() == TaskTag) {
                Q_ASSERT(task);
                task = task->parent();
                Q_ASSERT(task);
            }
        }
    }
}
```

这个方法用来从 XML 数据中读取一个任务,作为给定项的子项。这个方法使用递归创建该任务的所有直接和间接子项。在结构上,它的代码与先前在 `StandardTreeModel::load()` 方法中用到的代码是相同的。

通过传递根项作为参数,这个方法能用来加载一个完整的 XML 任务数据文件,但这里只用它来重新创建被拖动的项(包括所有直接和间接子项)作为指定项的直接子项。

我们现在已经介绍完了支持拖放操作的那些 `QAbstractItemModel` 方法。事实上,这些实现同样适用于列表和表格模型,因为 Qt 在所有的模型中使用了相同的方法来处理拖放问题,所以把这段代码稍做改变以让它直接工作在列表和表格模型下应该会比较简单的。

4.2.3.2.1 在树中移动项的方法

对于所有的项都是相同的类型、并且可任意嵌套的可编辑的树模型来说,提供拖放之外更多移动项的方法是有意义的。能够通过键盘来移动项特别受到那些不能或不想使用鼠标的人的欢

迎。当然,这里所讨论的所有移动方法都是通过 `QAction` 调用的,它们也能由用户使用鼠标点击相关的菜单项或工具栏按钮来触发。

这里提供了三组方法:在同级项之间进行上下移动的方法,剪切某个项并把它粘贴回树中的某个地方的方法,对项进行提升或降级操作的方法,即让一个项与它的父项同级或者成为它的某个同级项的子项。自然地,如同拖放操作的实现那样,这些方法不仅作用于被选中的项,还作用于这个项的所有直接和间接子项。

```
QModelIndex TreeModel::moveUp(const QModelIndex &index)
{
    if (!index.isValid() || index.row() <= 0)
        return index;
    TaskItem *item = itemForIndex(index);
    Q_ASSERT(item);
    TaskItem *parent = item->parent();
    Q_ASSERT(parent);
    return moveItem(parent, index.row(), index.row() - 1);
}
```

一个能向上移动的项,它前面至少要有一个同级的项,这就是说,它的行号必须大于0。如果是这样的话,我们调用 `moveItem()` 辅助方法,把这个项的父项指针、该项的当前行号(旧行号)和新行号作为参数传入——对于向上移动项来说,总是比原来的行号小一行。

`moveDown()` 方法(这里没有介绍)除了它必须至少要有一个在它的下方的同级项,并且新行号要比旧行号大一行之外,与 `moveUp()` 方法非常相似。

```
QModelIndex TreeModel::moveItem(TaskItem *parent, int oldRow,
                                int newRow)
{
    Q_ASSERT(0 <= oldRow && oldRow < parent->childCount() &&
              0 <= newRow && newRow < parent->childCount());
    parent->swapChildren(oldRow, newRow);
    QModelIndex oldIndex = createIndex(oldRow, 0,
                                       parent->childAt(oldRow));
    QModelIndex newIndex = createIndex(newRow, 0,
                                       parent->childAt(newRow));
    emit dataChanged(oldIndex, newIndex);
    return newIndex;
}
```

这个方法是被 `moveUp()` 和 `moveDown()` 方法调用来执行项移动操作的。`TaskItem::swapChildren()` 方法使用了 `QList::swap()` 方法来交换任务项的直接子项列表中的两个项。在执行完移动操作后, `moveItem()` 方法发射了两个 `dataChanged()` 信号来告知相关视图模型中的这两项发生了变化了,然后返回该项在新位置的模型索引。

一如既往,当调用 `createIndex()` 之时,传入了项的行号和列号(在这个特殊的模型中列号总是0),以及一个指向该项的 `TaskItem` 类型的指针作为参数。

图4.5演示了项的移动。事实上这个图可以当做把项A向下移动或把项B向上移动的一个演示,因为这两种移动操作的效果是一样的。图中右边显示的有阴影的项是受移动影响的部分:项A和项B,因为它们都被移动了,它们的父项即项P也是有阴影的,因为它的子项发生了变化。

这个方法,像许多移动相关的方法一样,向调用者返回一个模型索引。在大部分情况下,包括这里,这是指移

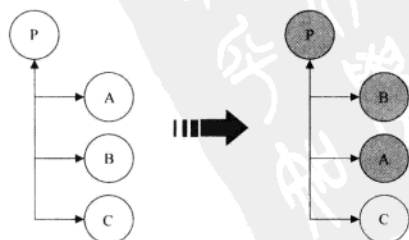


图4.5 通过与同级项的交换来向上或向下移动项

动项的模型索引。我们期望模型索引返回后,调用者会滚动到该项并把它选中。这对用户向上或向下移动项特别方便,因为用户可以选中一项,然后不停的调用 Up 动作(或按 Ctrl + Up 键或在 Mac OS X 平台下按⌘ + Up 键)来移动该项到每一个前面同级的项之前,直到它成为父项的第一个直接子项。相应地,用户也能调用 Down 动作(或按 Ctrl + Down 键或⌘ + Down 键)来把该项移动到它下方的同级的项之下,直到它成为父项的最后一个直接子项。

```
QModelIndex TreeModel::cut(const QModelIndex &index)
{
    if (!index.isValid())
        return index;
    delete cutItem;
    cutItem = itemForIndex(index);
    Q_ASSERT(cutItem);
    TaskItem *parent = cutItem->parent();
    Q_ASSERT(parent);
    int row = parent->rowOfChild(cutItem);
    Q_ASSERT(row == index.row());
    beginRemoveRows(index.parent(), row, row);
    TaskItem *child = parent->takeChild(row);
    endRemoveRows();
    Q_ASSERT(child == cutItem);

    if (row > 0) {
        --row;
        return createIndex(row, 0, parent->childAt(row));
    }
    if (parent != rootItem) {
        TaskItem *grandParent = parent->parent();
        Q_ASSERT(grandParent);
        return createIndex(grandParent->rowOfChild(parent), 0, parent);
    }
    return QModelIndex();
}
```

这个方法遵循大多数与移动操作相关的方法的共同模式:首先执行动作,然后创建一个模型索引返回给调用者,使得视图能够滚动到该项处并选择该项。

我们先销毁了 cutItem 指针指向的对象;如果它是 0 的话,这样做是没问题的,如果不是,那么删除(销毁)该项(包括它的所有直接和间接子项),因为它不可能被粘贴回任何地方了。然后获取剪切的任务项的指针,并把它保存在 cutItem 变量中;接着获取这个项的父项以及它在父项的直接子项列表中的行号。然后调用 beginRemoveRows() 方法通知模型/视图架构该项将被删除,然后从该剪切项的父项的直接子项列表中移除该项。这样 cutItem 所指向的剪切项现在没有父项了,我们有义务在合适的时候销毁(删除)它(事实上,它是通过三个方法来删除的:这是一个 clear() 还有 TreeModel 的析构函数)。一旦它被删除了,我们调用 endRemoveRows() 来通知模型/视图架构删除操作已经完成(进行行号比较、子项指针和剪切项指针的比較的 Q_ASSERT 调用,只是安全性测试而已)。

当从树中剪切掉一项后,任何关联的视图都会自动地把“最邻近的”一项作为当前项。如果剪切项的前面有同级项的话,那么视图将选择它前面同级的那一项作为当前项,如果失败了,那么看它后面是否有同级项,如果有,那么取它后面相邻的那一项作为当前项,如果再次失败了,那么取剪切项的父项作为当前项。我们倾向于总是选择前面的相邻的那一项,如果没有的话,那么选择父项,所以让这个方法返回我们想作为当前项的那个项的模型索引,并期待调用者把这个模型索引传递给自定义的 setCurrentIndex() 方法。如果对默认行为比较满意的话,就可以让这个方法返回 void 并且通过调用 endRemoveRows() 简单地结束该方法。在其他的与移动操作相关的方法中没有这样的机会——必须总是告诉相关的视图哪一项应该选中。

```

QModelIndex TreeModel::paste(const QModelIndex &index)
{
    if (!index.isValid() || !cutItem)
        return index;
    TaskItem *sibling = itemForIndex(index);
    Q_ASSERT(sibling);
    TaskItem *parent = sibling->parent();
    Q_ASSERT(parent);
    int row = parent->rowOfChild(sibling) + 1;
    beginInsertRows(index.parent(), row, row);
    parent->insertChild(row, cutItem);
    TaskItem *child = cutItem;
    cutItem = 0;
    endInsertRows();
    return createIndex(row, 0, child);
}

```

这个方法用来把一个项(及其所有直接和间接子项)粘贴回树中。我们选择总是把粘贴项作为当前选中项的新同级项插入,并占据它的父项的子项列表中新同级项下方的行。

这样做的一个结果就是用户永远不能粘贴某一项作为第一个子项——如此就必须先粘贴到第一个子项上(这样的话,粘贴项就成为了第二个子项)然后把粘贴项向上移动一行。另一方面我们能仅仅通过把一个项粘贴到最后一个子项上把它粘贴为最后一个子项。从另一方面讲,如果我们选择总是把粘贴项插入到当前项之前,用户就能粘贴某项作为第一个子项(粘贴到第一个子项上),但是却不能粘贴为最后一个子项,因为粘贴到最后一个子项上最终结果是放到了最后子项之前的那一项了。另外一种选择是弹出一个菜单(例如“粘贴到当前项之前”和“粘贴到当前项之后”选项)让用户来选择怎么做。

我们获得该项(将成为粘贴项的同级项),即当前选中的模型索引代表的那项。然后获取该项的父项,并找出该项在父项的直接子项列表中的行号。接着调用 `beginInsertRows()` 来告知模型/视图架构有一行将要被插入到模型中,并把剪切项插入到该项之后。

一旦粘贴完成后,设置 `cutItem` 为 0,因为不能再粘贴已经在树中的项了——只能粘贴从树中剪切出来的项。然后调用 `endInsertRows()` 告知模型/视图架构插入操作完成了,最后返回新粘贴项的模型索引以让视图能滚动到并选中该项。

```

QModelIndex TreeModel::promote(const QModelIndex &index)
{
    if (!index.isValid())
        return index;
    TaskItem *item = itemForIndex(index);
    Q_ASSERT(item);
    TaskItem *parent = item->parent();
    Q_ASSERT(parent);
    if (parent == rootItem)
        return index; // Already a top-level item

    int row = parent->rowOfChild(item);
    TaskItem *child = parent->takeChild(row);
    Q_ASSERT(child == item);
    TaskItem *grandParent = parent->parent();
    Q_ASSERT(grandParent);
    row = grandParent->rowOfChild(parent) + 1;
    grandParent->insertChild(row, child);
    QModelIndex newIndex = createIndex(row, 0, child);
    emit dataChanged(newIndex, newIndex);
    return newIndex;
}

```

提升某一项即把该项作为它的祖父项的一个直接子项,移动到它的前父项所在行之后。当然,该

项的所有直接和间接子项将随之一起提升。图 4.6 演示了项 B 的提升。阴影项即是受移动影响的那些项:项 B 的祖父项,即项 GP,成了它的新父项,项 B 原来的父项的子项列表中不再包含项 B。

我们先根据要提升的项的模型索引获得该任务项及它的父项。如果该项的父项是根项,那么该项已经是顶级项,不能再提升了,这时什么也不用做直接返回该项的模型索引即可。否则,找到该项在父项的直接子项列表中的行号并且把它从父项的直接子项列表中移除(这时它就变得没有了父项并且是无所属的,因为 `TaskItem::takeChild()` 方法把它从父项的直接子项列表中去除掉了,并且把它的父项设置为 0)。

我们进行了安全的 `Q_ASSERT` 断言测试以确保从当前项的父项中获取的子项就是要提升的项。然后获取它的祖父项,并找到父项在它自己的父项的直接子项列表中的位置。接着,在原父项之后的行插入该项;`TaskItem::insertChild()` 方法会重新设置插入项的父项索引属性,所以最后,这个项安全地返回到了树中,并且有一个正确的父项索引属性。

在最后,程序为该提升的项创建了一个模型索引并发射了一个 `dataChanged()` 信号给相关的视图告知模型已经发生了变化了。最后返回提升项的模型索引给调用者。

```
QModelIndex TreeModel::demote(const QModelIndex &index)
{
    if (!index.isValid())
        return index;
    TaskItem *item = itemForIndex(index);
    Q_ASSERT(item);
    TaskItem *parent = item->parent();
    Q_ASSERT(parent);
    int row = parent->rowOfChild(item);
    if (row == 0)
        return index; // No preceding sibling to move this under
    TaskItem *child = parent->takeChild(row);
    Q_ASSERT(child == item);
    TaskItem *sibling = parent->childAt(row - 1);
    Q_ASSERT(sibling);
    sibling->addChild(child);
    QModelIndex newIndex = createIndex(sibling->childCount() - 1, 0,
                                       child);
    emit dataChanged(newIndex, newIndex);
    return newIndex;
}
```

对某项进行降级意味着将移动该项使它成为它前面同级项的直接子项。当然,要进行降级的项的所有直接和间接子项也会随之一起移动。我们可以降级该项到它的前面的同级项(降级后就变成该项的父项了)的直接子项列表中的任何地方,但我们选择总是把降级项放到前一个同级项的最后一个直接子项后面。图 4.7 演示了项 B 的降级。阴影项是受移动影响的项:项 B 的前一个同级项 A,成了项 B 的新父项,项 B 成了项 A 的最后一个直接子项,项 B 从它原来的父项 P 的直接子项中移除。

进行降级操作时,首先获取要降级的项的模型索引对应的任务项及它的父项。如果该项是它的父项的第一个子项,那么它上面没有同级项作为移动动作的目的地,所以什么也不用做直接返回该项的模型索引即可。否则,把它从它的父项中移除——此时它是无父项且无所属的,因为 `TaskItem::takeChild()` 方法把它从它的父项的直接子项列表中移除并设置它的父项为 0。

使用如同提升某项相似的方式,我们也用 `Q_ASSERT` 做断言检查来确保从父项中获取的子项就是要降级的项。然后获取该项在父项的直接子项列表中的前一个同级项并把它作为这个同级

项的最后一个子项添加进去。`TaskItem::addChild()`方法将重新设置新添加的项的父项模型索引属性,所以调用完这个方法后,要降级的项已经安全地返回到了树中,并有了正确的父项模型索引属性。

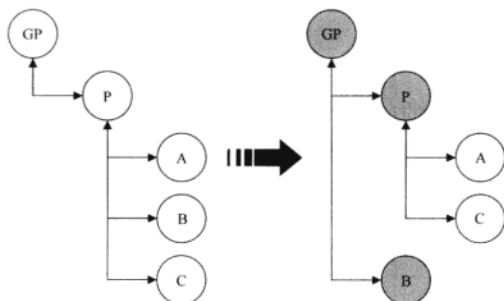


图 4.6 提升某项为它祖父项的直接子项

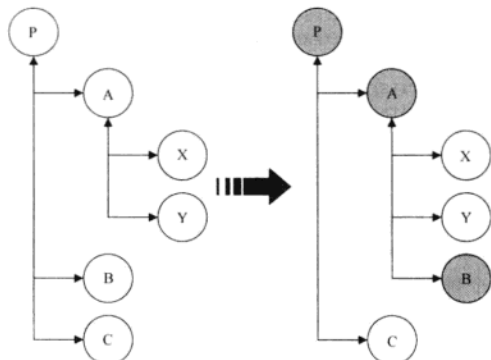


图 4.7 降级某项使它成为它上一个同级项的最后一个直接子项

在末尾,如在 `promote()` 方法中所做的,为该降级项创建了一个模型索引,并发射了一个 `data-Changed()` 信号来通知相关视图模型发生了变化了。最后,返回该降级模型索引给调用者。

现在我们已经介绍了添加到 `QAbstractItemModel` API 中的那些在树中移动项的所有方法。假设表示项的类使用了一个子项列表并且有那些在这里使用的方法 (`addChild()`、`insertChild()`、`takeChild()`,等等,或者等同的方法),上面介绍的那些实现应该非常容易就能重用到其他自定义树模型中。这种复用性是不错的,因为能通过那些 `Q_ASSERT` 调用所留下的线索做出可能的诊断。尽管大部分方法看起来简单,但想使用正确就还需要一些技巧。

4.2.3.3 保存和加载树中项的方法

在这一小节中将介绍支持从文件中加载树中各项以及把树中各项的数据保存到文件所需要的方法,还包括处理任务路径的方法。尽管 `timelog2` 应用程序比 `timelog1` 多了许多功能(特别是支持了拖放、复制和粘贴和移动项),两个应用程序使用的是相同的 XML 格式的文件,所以它们的文件是完全可互换的。

```
void TreeModel::clear()
{
    delete rootItem;
    rootItem = 0;
    delete cutItem;
    cutItem = 0;
    timedItem = 0;
    reset();
}
```

支持 `fileNewAction` 需要这个方法。我们必须删除根项和剪切项,因为我们对其有所有权;但是计时项(正在执行任务对应的项)是在树中的所有归根项(不需要对计时项多做介绍了,因为它唯一的价值在于做任务计时,在我们关心的模型/视图编程上下文中并不重要)。最后调用了 `QAbstractItemModel::reset()`,告知任何相关的视图模型数据已经彻底发生了变化,所以它们必须重新获取每一项要显示的内容。

```

void TreeModel::save(const QString &filename)
{
    ...
    QDomStreamWriter writer(&file);
    writer.setAutoFormatting(true);
    writer.writeStartDocument();
    writer.writeStartElement("TIMELOG");
    writer.setAttribute("VERSION", "2.0");
    writeTaskAndChildren(&writer, rootItem);
    writer.writeEndElement(); // TIMELOG
    writer.writeEndDocument();
}

```

这个方法的开头与 `StandardTreeModel::save()` 方法是相同的(所以省略了这一段代码),使用已存在的文件名或传入的文件名参数,并且在文件不能打开的情况下抛出异常。所有的工作都由 `writeTaskAndChildren()` 方法完成,就是我们先前在介绍拖放功能时介绍过它的实现,并且序列化一个项以及它的所有子项为 XML 格式而用到的方法。

```

void TreeModel::load(const QString &filename)
{
    ...
    clear();
    rootItem = new TaskItem;
    QDomStreamReader reader(&file);
    readTasks(&reader, rootItem);
    if (reader.hasError())
        throw AQP::Error(reader.errorString());
    reset();
}

```

这个方法开始部分与 `save()` 方法的开始部分相似,是关于文件名的处理并在文件无法打开的情况下抛出异常,所以再次省略了这段代码。在清除掉现存任务后创建了一个新根项,并且使用 `readTasks()` 方法来生成整个树,以前面创建的根项为根,并且从给定的 `QDomStreamReader` 对象中获取数据。这个 `readTasks()` 方法与先前使用过的那个在树中指定的父项下重新创建拖放数据的 `readTasks()` 方法一样。

```

QStringList TreeModel::pathForIndex(const QModelIndex &index) const
{
    QStringList path;
    QModelIndex thisIndex = index;
    while (thisIndex.isValid()) {
        path.prepend(data(thisIndex).toString());
        thisIndex = thisIndex.parent();
    }
    return path;
}

```

这个方法提供了一个 `QStringList` 对象来代表树中的一个特定项,它使用了与 `StandardTreeModel` 中的同名方法相同的逻辑。

它首先把给定项(通过参数 `index` 所对应的项)的文本加到 `QStringList` 对象 `path` 中,接着把它的父项的文本插入到 `QStringList` 的最前面,然后把祖父项的文本插入到 `QStringList` 的最前面,如此这般直到最顶层。要记住顶级项索引的父项模型索引不是根项(它是顶级项的父项)而是一个无效的模型索引。

这个方法也在主窗口的关闭事件处理中用到了,在应用程序退出时使用 `QSettings` 对象保存当前选中项。

```

QModelIndex TreeModel::indexPathForPath(const QStringList &path) const
{
    return indexPathForPath(QModelIndex(), path);
}

```

```

QModelIndex TreeModel::indexForPath(const QModelIndex &parent,
                                     const QStringList &path) const
{
    if (path.isEmpty())
        return QModelIndex();
    for (int row = 0; row < rowCount(parent); ++row) {
        QModelIndex thisIndex = index(row, 0, parent);
        if (data(thisIndex).toString() == path.at(0)) {
            if (path.count() == 1)
                return thisIndex;
            thisIndex = indexForPath(thisIndex, path.mid(1));
            if (thisIndex.isValid())
                return thisIndex;
        }
    }
    return QModelIndex();
}

```

上面这两个方法与 `pathForIndex()` 相反——它们接受一个模型索引和一个任务路径,返回任务路径对应的模型索引。公有方法接受一个任务路径参数,它调用了私有方法,并把一个无效的模型索引(在 Qt 的模型/视图架构中,它特指根)作为父项模型索引当做一个参数,把给定的任务路径作为另一个参数。

私有方法遍历父模型索引的直接子项模型索引列表来查找看哪一个子项的文本与任务路径中的第一个文本相同。如果找到了,那么该方法递归地调用自身,并把找到的直接子项的模型索引作为新的父项模型索引当做一个参数传递过去,把任务路径中去除了第一项(第一项已经找到匹配项了)后余下的路径作为第二个参数传递过去。最后要么所有的文本都会找到对应的匹配项的模型索引并返回,否则匹配失败就会返回一个无效的模型索引。

在 *Timelog* 应用程序启动时,它使用 `QSettings` 来获取上次使用的文件路径以及该文件中选中项的任务路径。如果有这样的文件,那么 `indexForPath()` 方法会在加载完文件后用来查找指定任务路径对应项的模型索引,这样就能滚动到该项所在之处并把它选中,把树恢复到上次关闭该应用程序时的相同状态。

事实上状态可能不能完全恢复,因为我们只是展开了树以使得它足以显示选中项,然而用户可能在上次结束应用程序的时候还留有对树中其他位置的展开。应该尽可能地恢复树的所有状态,但如果用户有大量的完全展开的并且带有大量子项的顶级项,这样做可能会在 `QSettings` 对象中耗费大量的存储(例如在 Windows 注册表中),所以如果确实需要的话,或许在数据文件中保存记录的状态是个更好的方法(例如像 `expanded` 或 `visible` 的属性),或者使用另一个文件来保存,因为,举例来说,Windows 注册表是有尺寸限制的。

现在我们已经完成了对树模型的介绍,了解了如何创建一个自定义 `QStandardItemModel` 来保存树中的项。此外,我们还了解到了如何创建一个自定义的 `QAbstractItemModel`,提供与其他模型相同的 `QAbstractItemModel` API,加上那些支持在树中移动项,支持从磁盘中加载和保存数据项到磁盘的那些扩展 API。在下一章中我们把注意力转向自定义委托对象,在关于模型/视图相关的四章内容的最后一章我们将介绍自定义视图。

第5章 模型/视图委托

- 与数据类型相关的编辑器
- 与数据类型相关的委托
- 与模型相关的委托

本章将介绍模型/视图的委托,假设读者如第3章开始时所述已基本熟悉了 Qt 的模型/视图架构。

所有的 Qt 标准视图类(QListView、QTableView、QColumnView、QTreeView 和 QComboBox)都要访问的数据进行显示和编辑(对于可编辑模型)提供了一个 QStyledItemDelegate^①。

在本章中将介绍如何设置和使用自定义委托对象,这些委托对象能让我们对视图中显示的项的外观进行完全的控制,或者允许对可编辑项提供自己的编辑器,或者二者兼具。广义地讲,有三种使用委托的方式,在这一章中都将进行介绍。

Qt 的内置委托对象使用特定的窗口部件来对特定的数据类型进行编辑。在 5.1 节将介绍如何把 Qt 使用的默认窗口部件改变为自己选择的其他内置窗口部件或自定义窗口部件。这是一种非常强大的方法——它将影响所有视图中的相关数据类型的可编辑项——但因为这个原因,它也是最不灵活的方法,尤其是与使用自定义委托相比时。

在 5.2 节将介绍如何创建可以应用于特定行或列的自定义数据类型相关(datatype-specific)的委托。这些委托非常通用,能够在各种模型中重复使用。而且不同于简单地更改编辑部件,创建自定义委托允许我们同时控制项的显示和编辑。这里将介绍这种类型的两个示例,第一个示例是一个简单的只读委托(read-only delegate),它以自定义方式显示日期时间类型数据。第二个示例是一个更加复杂的委托,它能够显示和编辑像在前一章中的 Timelog 示例中用到的富文本项。

在一些情况下,使用自定义模型相关(model-specific)的委托来处理模型中所有的项要比在模型的行或列中使用数据类型相关的委托要更加方便。在 5.3 节中将介绍如何实现一个模型相关的委托,以第3章中的 Zipcodes 示例中使用的委托为例。

5.1 与数据类型相关的编辑器

如果想提供一个仅仅取决于项类型的全局编辑器,可以创建一个 QItemEditorFactory 对象,并且注册一个特定的窗口部件来作为该特定类型(或若干个特定类型)的编辑器。例如:

```
QItemEditorFactory *editorFactory = new QItemEditorFactory;
QItemEditorCreatorBase *numberEditorCreator = new
    QStandardItemEditorCreator<SpinBox>();
editorFactory->registerEditor(QVariant::Double,
                             numberEditorCreator);
editorFactory->registerEditor(QVariant::Int,
                             numberEditorCreator);
QItemEditorFactory::setDefaultFactory(editorFactory);
```

^① Qt 还有一个 QItemDelegate 类,但从 Qt 4.4 起就由 QStyledItemDelegate 取代了。

上面这段代码的意思是,在应用程序中的所有视图中,其所有值为双精度型(double)或整型(int)的可编辑项使用自定义的 SpinBox 部件作为编辑器。

```
explicit SpinBox(QWidget *parent=0)
    : QDoubleSpinBox(parent)
{
    setRange(-std::numeric_limits<double>::max(),
             std::numeric_limits<double>::max());
    setDecimals(3);
    setAlignment(Qt::AlignVCenter|Qt::AlignRight);
}
```

我们只重新实现了构造函数。设置微调框的取值范围为系统支持的双精度型的最小负数到最大正数,并且显示三位小数,右对齐显示^①。这个编辑器在图 5.1 中有显示。注册了这个编辑器后就能确保在应用程序中的所有视图中所有的双精度型和整型的项的显示和编辑是一致的。

	25	26	27	28	29	30	31
21	72359.810	97109.063	56523.181	114758.495	-7939.302	129745.586	27442.665
22	31813.620	35587.706	120365.294	-6052.444	167123.596	109102.557	135412.225
23	-8442.069	85859.183	135574.319	157860.401	22872.170	89297.041	
24	128915.849	127434.076	40193.191	142984.805	93907.571	141220.210	-5752.025
25	174512.228	174400.054	8151.157	17034.942	-4152.936	167356.952	81396.061
26	-4452.230	112293.587	161327.830	140286.507	5952.576	137362.149	189036.464

图 5.1 注册 SpinBox 后的编辑状态

遗憾的是,像这样注册的编辑器和 QStandardItem 中的那些非字符串类型的数据不能很好地一起工作——至少不能直接使用。这是因为在 QStandardItem 项中是以 QString 类型来保存数据的,所以当开始编辑时供编辑的数据是 QString 类型的。这就不能触发使用注册的编辑器(SpinBox),因为它只注册为双精度型和整型数据使用。这种情况的解决方案很简单:只要确保从 QStandardItemModel 中存储和获取数据时使用 Qt::EditRole,并且使用 QStandardItem 的派生类来保存数据。下面就是这样的一个保存双精度数据的派生类示例:

```
class StandardItem : public QStandardItem
{
public:
    explicit StandardItem(const double value) : QStandardItem()
    { setData(value, Qt::EditRole); }

    QStandardItem *clone() const
    { return new StandardItem(data(Qt::EditRole).toDouble()); }

    QVariant data(int role=Qt::UserRole+1) const
    {
        if (role == Qt::DisplayRole)
            return QString("%1").arg(QStandardItem::data(Qt::EditRole)
                                     .toDouble(), 0, 'f', 3);
        if (role == Qt::TextAlignmentRole)
            return static_cast<int>(Qt::AlignVCenter|Qt::AlignRight);
        return QStandardItem::data(role);
    }
};
```

在构造函数中使用 Qt::EditRole 角色来存储双精度数据。还提供了一个 clone() 方法,以确保如果模型复制一个项的话它能正确地创建一个 StandardItem 实例而非普通的 QStandardItem 实例。

^① 我们不能使用 std::numeric_limits<T>::min(), 因为对于浮点型数据来说,它将返回一个大于 0 的最小值(小数),而不是最小的负数(对整型则返回最小负数)。

`QStandardItem::data()`方法与`QAbstractItemModel::data()`方法不同;具体来说,它是使用了基类版本的传统 C++ 方法,所以对于未处理的情况,它应该总是返回调用基类的返回值——而不是无效的 `QVariant`。这里只需要处理显示和文本对齐角色以获得想要的格式,双精度数据将会通过调用基类方法(通过 `Qt::EditRole`)以 `QVariant` 变量的形式正确返回。

上面的这几段展示如何使用一个项的编辑器生成器的代码段是从第 7 章的数字表格(`Number Grid`)示例(`numbergrid`)中摘出来的。

为特定数据类型的项注册一个编辑器部件是非常强大的功能,因为它能对一个应用程序产生全局影响。在实践中却很少这样做,我们更乐于一个模型一个模型地定制项的外观和行为,因为正如后面两节要介绍的,可以通过自定义委托非常容易地获得这样的效果。

5.2 与数据类型相关的委托

如果我们创建了大量的自定义模型,通常要为每一个自定义模型创建一个自定义委托,可能会发现最终有大量的重复代码。例如,可能有几个模型,每一个都与一个自定义委托关联。因为每个模型在不同的列都有不同的数据类型,所以这些委托都会有所不同,尽管处理每一个特定类型的代码(例如,为日期列使用自定义的日期编辑器)是相同的。

一个避免重复和提高代码复用性的方法(不仅仅在一个应用程序中,而是在不同的应用程序中)是为特定的行或列创建数据类型相关的委托。例如,如果创建了一个针对日期类型的委托,就可以为有一个或几个日期列的模型设置这个列委托。这将消除大量的重复代码,并且能使在包含日期列的新模型中使用这个委托的工作变得轻而易举。我们还可以创建其他数据类型相关的委托,例如,为角度、颜色、货币、时间,等等,所有的这些都能避免代码重复并确保在不同的应用程序中的一致性。

在这一节中将介绍这样的两个委托,它们都是用于特定列的,当然也可以方便地应用于特定行。第一个示例是一个简单的只读委托,用于以自定义方式显示日期时间,第二个示例是一个显示和编辑“富文本”——即用一个 HTML 子集来标记文字的加粗、斜体、颜色等格式数据的委托。这个委托在前一章中的 `Timelog` 应用程序使用到了。

5.2.1 一个只读的列或行委托

图 5.2 所示的是文件夹查看器(`Folder View`)应用程序(`folderview`),它有两个 `QTreeView` 部件,右边的 `QTreeView` 部件使用了一个自定义的 `DateTimeDelegate` 委托把时间数据显示为一个时钟,日期数据显示为 ISO 8601 格式的字符串。获取数据使用的模型是 `QFileSystemModel`,为了正好能在一页中显示截图所示内容,我们隐藏了两列。

`DateTimeDelegate` 把最后修改时间显示为一个模拟时钟面板,同时使用浅色背景表示上午(AM)时间,深色背景表示下午(PM)时间——并用褪色的背景表示日期早于今天。

在介绍委托的代码之前,先简单地看一下创建模型、视图和委托的代码,以了解来龙去脉。

```
QFileSystemModel *model = new QFileSystemModel;
model->setRootPath(QDir::homePath());
QModelIndex index = model->index(QDir::homePath());
QTreeView *view = new QTreeView;
view->setItemDelegateForColumn(3, new DateTimeDelegate);
view->setModel(model);
view->setColumnHidden(1, true);
view->setColumnHidden(2, true);
view->scrollTo(index);
view->expand(index);
view->setCurrentIndex(index);
```

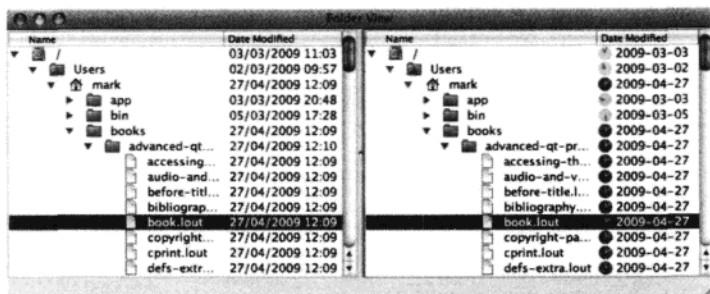


图 5.2 只读的 DateTimeDelegate 委托(右边的视图)

静态方法 `QDir::homePath()` 返回当前用户的主目录(home directory)。 `QDir` 类还有其他相似的方法,包括 `QDir::currentPath()`、`QDir::rootPath()` 和 `QDir::tempPath()`,它们都返回字符串。该类还有相应的返回 `QDir` 对象的方法(`QDir::home()`, 等等)。

QFileSystemModel 是可编辑的,所以它可以用做文件管理器的基础。这里仅仅用它来为 QTreeView(我们对它的第 4 列设置了 QDateTimeDelegate 委托)提供数据,其他所有列将使用树形视图的内置委托 QStyledItemDelegate。QFileSystemModel::setRootPath() 的调用并不是设置当前选中项(setCurrentIndex() 才是),而是设置由 QFileSystemWatcher 监视的目录。如果被监视的目录下的文件或子目录发生任何更改,这些更改(例如一个文件被删除或更新了)就会反馈到模型中,然后在与之相关的视图上显示出来。

对于 `DateTimeDelegate`,需要重新实现构造函数和 `paint()` 方法,因为我们只想改变它要处理的项的外观,而不是行为。

```
class DateTimeDelegate : public QStyledItemDelegate
{
    Q_OBJECT

public:
    explicit DateTimeDelegate(QObject *parent=0)
        : QStyledItemDelegate(parent) {}

    void paint(QPainter *painter, const QStyleOptionViewItem &option,
               const QModelIndex &index) const;

private:
    ...
};
```

构造函数只是简单调用了基类的构造函数。下面看一下 `paint()` 方法以及它使用的一些私有的辅助方法,看看项是如何渲染的。

```
void QDateTimeDelegate::paint(QPainter *painter,
    const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    const QFileSystemModel *model =
        object_cast<const QFileSystemModel*>(index.model());
    Q_ASSERT(model);
    const QDateTime &lastModified = model->lastModified(index);
    painter->save();
    painter->setRenderHints(QPainter::Antialiasing|
        QPainter::TextAntialiasing);

    if (option.state & QStyle::State_Selected)
        painter->fillRect(option.rect, option.palette.highlight());
    const qreal diameter = qMin(option.rect.width(),
        option.rect.height());
```

```

const QRectF rect = clockRect(option.rect, diameter);
drawClockFace(painter, rect, lastModified);
drawClockHand(painter, rect.center(), diameter / 3.5,
    ((lastModified.time().hour() +
    (lastModified.time().minute() / 60.0)) * 30));
drawClockHand(painter, rect.center(), diameter / 2.5,
    lastModified.time().minute() * 6);
drawDate(painter, option, diameter, lastModified);
painter->restore();
}

```

首先获取指定项(可能是文件、目录或文件系统对象)的最后修改日期/时间以进行绘制。然后保存绘制对象(painter)的状态并且启用反走样(antialiasing)^①。

如果该项是选中的,就从调色板中挑选适当的高亮背景色来绘制背景。然后计算时钟面板的直径(后面的各种计算都要用到这个值)和用来显示时钟面板的矩形区域。

一切都准备好后,开始绘制时钟面板、分针、时针以及日期(文字方式),最后恢复绘制对象的状态为开始绘制之前的状态,以便于它能用于下一项的绘制。

```

QRectF DateTimeDelegate::clockRect(const QRectF &rect,
    const qreal &diameter) const
{
    QRectF rectangle(rect);
    rectangle.setWidth(diameter);
    rectangle.setHeight(diameter);
    return rectangle.adjusted(1.5, 1.5, -1.5, -1.5);
}

```

这个方法基于给定的矩形区域参数返回一个矩形,保持相同的 x 和 y 坐标,但根据给定的直径缩小为一个正方形,最后再缩小一些以获得一些空白边距。

`QRect::adjusted()` 方法(以及 `QRectF` 类的同名方法)返回一个新的矩形区域,这个矩形区域坐标的左上角和右下角按照给定的值进行调整。所以在这个例子中,左上角坐标向右、向下移动了(因为 y 坐标是向下增长的)1.5 个像素,右下角则向左、向上移动了 1.5 个像素。

```

void DateTimeDelegate::drawClockFace(QPainter *painter,
    const QRectF &rect, const QDateTime &lastModified) const
{
    const int Light = 120;
    const int Dark = 220;
    int shade = lastModified.date() == QDate::currentDate()
        ? Light : Dark;
    QColor background(shade, shade,
        lastModified.time().hour() < 12 ? 255 : 175);
    painter->setPen(background);
    painter->setBrush(background);
    painter->drawEllipse(rect);
    shade = shade == Light ? Dark : Light;
    painter->setPen(QColor(shade, shade,
        lastModified.time().hour() < 12 ? 175 : 255));
}

```

这个方法用于绘制时钟面板,即画一个椭圆形以产生一个圆(如果矩形区域是如同本例中的正方形的话)。大部分代码是与使用 RGB(Red 红、Green 绿、Blue 蓝)值的颜色相关的,这些值必须在 0 ~ 255 范围内,而在本例中取决于时间是上午(AM)还是下午(PM)以及日期是今天还是更早一些。最后设置用来绘制时钟的时针和分针的画笔的颜色,因为它们是与时钟面板的颜色相关的(以确保较好的对比度)。

^① 在本书写作时,Qt 的 `QPainter` 的文档并没有指出默认打开的渲染提示属性是什么,所以这里使用最保险的方式,总是指定我们想要的渲染属性。

```
void DateTimeDelegate::drawClockHand(QPainter *painter,
    const QPointF &center, const qreal &length,
    const qreal &degrees) const
{
    const qreal angle = AQP::radiansFromDegrees(
        (qRound(degrees) % 360) - 90);
    const qreal x = length * std::cos(angle);
    const qreal y = length * std::sin(angle);
    painter->drawLine(center, center + QPointF(x, y));
}
```

这个方法被调用了两次,一次用以绘制分针,一次用以绘制时针。圆心是时钟面板的中心,长度要比半径小些(时针要比分针更短一些),角度则由指针(分针、时针)所代表的相应时间按比例计算出来。我们要确保角度值在范围之内,并且减去 90°,这样 0°的位置从坐标系统中使用的东边(East)移动到模拟时钟中使用的北边(North)。然后计算出指针终点的位置并从中心到终点画一条线(qRound()函数在表 1.2 中有介绍。AQP::radiansFromDegrees()函数在本书的 aqp. {hpp, cpp} 模块中)。

```
void DateTimeDelegate::drawDate(QPainter *painter,
    const QStyleOptionViewItem &option, const qreal &diameter,
    const QDateTime &lastModified) const
{
    painter->setPen(option.state & QStyle::State_Selected
        ? option.palette.highlightedText().color()
        : option.palette.windowText().color());
    QString text = lastModified.date().toString(Qt::ISODate);
    painter->drawText(option.rect.adjusted(
        qRound(diameter * 1.2), 0, 0, 0), text,
        QTextOption(Qt::AlignVCenter|Qt::AlignLeft));
}
```

我们使用窗口文本的颜色来绘制日期,如果该项是被选中的,那么就使用高亮文本颜色。同时把 x 坐标向右移动,留出时钟面板和空白边距(margin)的空间。

QTextOption 类用来存储一段富文本的对齐方式(alignment)、换行模式(wrap mode)、制表符位置(tab stop)以及其他各种格式标记。在绘制文字时最常用的是提供想要的文本对齐方式以及作用于多行文本的折行模式。

像这里所做的这样,通过创建自定义委托并重新实现它的 paint() 方法,可以容易的实现以自定义方式绘制日期时间或其他类型的模型数据。并且如同 Qt 中的大多数绘制一样,这部分工作主要涉及到使用合适的颜色以及尺寸和位置的数学运算。但是对于可编辑项,我们可能更倾向于提供自己的编辑部件,这需要重新实现如下一小节所介绍的更多方法。

5.2.2 一个可编辑的列或行委托

一个自定义委托可以用来渲染(render)项或编辑项,或二者兼具。对于渲染项,只需要重新实现 paint() 方法即可,但如果想支持编辑,就必须实现表 5.1 列出来的那些 QStyledItemDelegate API——至少实现 createEditor()、setEditorData() 和 setModelData() 方法。

基类的 sizeHint() 和 updateEditorGeometry() 实现几乎总是够用了,因此很少需要重新实现它们。同样地,通常也不需要重新实现 paint() 方法,特别是如果数据只是简单的纯文本、日期、时间或数字的话。

在第 4 章的 Timelog 应用程序中我们见到过“富文本”(一个简单的 HTML 子集,用以支持基本的文字效果,比如加粗、斜体和颜色)的使用。此外,富文本项(任务项)是可复选的。为了支持这样的需求,我们创建了一个 RichTextDelegate,在图 5.3 中所示,展示了它显示上下文菜单时的样子。

表 5.1 QStyledItemDelegate 的 API

方 法	说 明
createEditor(parent, styleOption, index)	为指定模型索引 index 所对应的项创建一个合适的编辑用的窗口部件并返回
paint(painter, styleOption, index)	绘制给定模型索引 index 所对应的项(对于纯文本、日期或数字类型很少有需要去重新实现)
setEditorData(editor, index)	使用给定模型索引 index 所对应的模型项的数据来填充编辑器 editor
setModelData(editor, model, index)	从编辑器 editor 中获取数据并设置为给定模型索引 index 所对应的模型项的数据
sizeHint(styleOption, index)	返回委托需要的、显示或编辑给定模型索引 index 所代表的项的尺寸
updateEditorGeometry(editor, styleOption, index)	为就地编辑(in-place editing)设置编辑器 editor 的尺寸和位置(很少需要重新实现)

RichTextDelegate 对于渲染和编辑的功能都提供了。这个类实现了 QStyledItemDelegate API 中的大多数方法——具体为 paint()、sizeHint()、createEditor()、setEditorData() 和 setModelData()。此外,它还有一个私有的槽 closeAndCommitEditor()和两项私有数据成员(一个 QCheckBox 指针和一个 QLabel 指针)。我们将介绍所有这些方法,先从构造函数开始。

```
RichTextDelegate::RichTextDelegate(QObject *parent)
    : QStyledItemDelegate(parent)
{
    checkbox = new QCheckBox;
    checkbox->setFixedSize(
        qRound(1.3 * checkbox->sizeHint().height()),
        checkbox->sizeHint().height());
    label = new QLabel;
    label->setTextFormat(Qt::RichText);
    label->setWordWrap(false);
}
```

在自定义委托中渲染项的时候,可以采取三种方法。一种是自己绘制每一项——在 DateTime 委托中,就是这样做的。它的缺点就是必须自己处理平台间的差异。另一种是使用 Qt 中的 QStyle 类,例如使用 QStyle::drawControl()、QStyle::drawComplexControl(),等等(一种强大的,但比较底层的实现方式,需要十分小心并且需要相当多的代码)。这里我们选择了简单一些的比较高层次的方式:绘制部件,在这里是复选框和一个窗口部件,留给 Qt 去处理平台相关性,保持代码尽可能简洁。

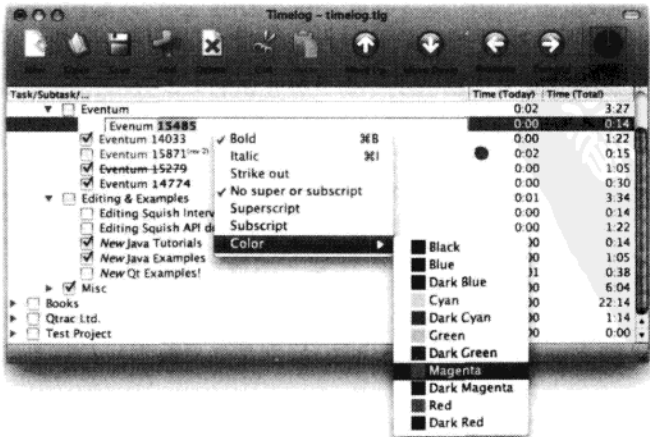


图 5.3 RichTextDelegate 委托显示效果

如果只是想渲染为纯文本格式,那么只需要一个复选框部件就可以了,因为复选框部件将显示一个复选框和一个关联的文本内容。但是由于我们想显示富文本,所以复选框部件仅仅用来显示复选框(设置它的文本为空),而使用一个标签部件来显示富文本内容。

我们设置复选框占据的宽度比它实际需要的多一点,以提供一些边距,这样它就不会紧靠着标签部件了。然后设置标签部件把内容以富文本(HTML)格式显示并且不做任何单词折行。

```
-RichTextDelegate() { delete checkbox; delete label; }
```

当委托销毁时,必须删除掉复选框部件和标签部件——这个析构函数作为内联形式直接写到了头文件里。

为易于解说,我们把 paint() 方法分成 4 部分。

```
void RichTextDelegate::paint(QPainter *painter,
    const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    bool selected = option.state & QStyle::State_Selected;
    QPalette palette(option.palette);
    palette.setColor(QPalette::Active, QPalette::Window,
        selected ? option.palette.highlight().color()
            : option.palette.base().color());
    palette.setColor(QPalette::Active, QPalette::WindowText,
        selected
            ? option.palette.highlightedText().color()
            : option.palette.text().color());
```

首先基于参数 option(QStyleOptionViewItem 类型)的调色板创建一个新的调色板(palette),然后设置新调色板的 QPalette::Window(背景)色和 QPalette::WindowText(前景,即文本)色为从 option 参数中获得的颜色,并且把该项是否选中计入考量(选择使用 QPalette::base() 颜色而非 QPalette::window() 颜色作为背景色,这是因为在我们的测试机上它有更好的跨平台性)。

```
int yOffset = checkbox->height() < option.rect.height()
    ? (option.rect.height() - checkbox->height()) / 2 : 0;
QRect checkboxRect(option.rect.x(), option.rect.y() + yOffset,
    checkbox->width(), checkbox->height());
checkbox->setPalette(palette);
bool checked = index.model()->data(index, Qt::CheckStateRole)
    .toInt() == Qt::Checked;
checkbox->setChecked(checked);
```

首先,进行复选框相关的设置。先创建一个将在后面绘制复选框时用到的 checkboxRect 矩形区域对象。创建了该矩形之后,如果 option 矩形的高度大于复选框所需高度的话,那么设置该矩形在可用空间中垂直居中。接着设置复选框的调色板为前面创建的调色板对象,最后设置复选框的选中状态为对应项的选中状态。

```
QRect labelRect(option.rect.x() + checkbox->width(),
    option.rect.y(), option.rect.width() - checkbox->width(),
    option.rect.height());
label->setPalette(palette);
label->setFixedSize(qMax(0, labelRect.width()),
    labelRect.height());
QString html = index.model()->data(index, Qt::DisplayRole)
    .toString();
label->setText(html);
```

尽管我们已经在构造函数中设置了复选框为固定大小的,并且永远不会改变它,但对于标签(尽管也设置了固定大小了)就必须为每一项进行大小设置。为标签创建的 labelRect 矩形区域是基于参数 option 给定的矩形区域的,只是向右边进行了偏移(宽度减少了 offset),以空出复选框的空

间。宽度的减少可能会导致产生负的宽度(例如假如用户把所在的窗口的宽度减少到了足够程度),所以我们使用 `qMax()` 来进行纠正。一旦标签设置了调色板和大小尺寸,就获取对应项的文本(HTML 格式的)并把它设置为标签的文本。

这时候,复选框和标签都有了正确的调色板、大小尺寸和内容,我们也有了进行绘制所需要的矩形区域。

```
QString checkboxKey = QString("CHECKBOX:%1.%2").arg(selected)
    .arg(checked);
paintWidget(painter, checkboxRect, checkboxKey, checkbox);
QString labelKey = QString("LABEL:%1.%2.%3x%4").arg(selected)
    .arg(html).arg(labelRect.width()).arg(labelRect.height());
paintWidget(painter, labelRect, labelKey, label);
}
```

我们已经把绘制窗口部件的过程提取到了私有辅助方法 `paintWidget()` 中。而且使用 Qt 的全局 `QPixmapCache` 对象省掉一遍又一遍对相同图像的重新绘制。该缓存使用一个字符串来标志每个图像——我们使用选择状态和内容(复选框的选中状态和标签的文本内容)进行标志。所以,对于复选框来说,在缓存中最多有四个图像:(选中的、未复选的)、(选中的、复选的)、(未选中的、未复选的)和(未选中的、复选的)。一旦有了用于缓存的标志字符串,就调用 `paintWidget()` 方法。

```
void RichTextDelegate::paintWidget(QPainter *painter,
    const QRect &rect, const QString &cacheKey,
    QWidget *widget) const
{
    QPixmap pixmap(widget->size());
    if (!QPixmapCache::find(cacheKey, &pixmap)) {
        widget->render(&pixmap);
        QPixmapCache::insert(cacheKey, pixmap);
    }
    painter->drawPixmap(rect, pixmap);
}
```

先创建一个指定尺寸的空白图像。`QPixmapCache::find()` 方法用来根据指定的标志从缓存中获取一个图像。如果在缓存中找到该标志,那么该方法返回 `true`,并且填充通过指针(或对于 Qt 4.5 或更早的版本使用常量引用)传入的 `QPixmap` 对象;否则返回 `false`。所以第一次请求一个特定标志的图像时,该标志是找不到的,我们把给定的窗口部件渲染到空白图像上,并把它插入到缓存中。最后,在给定的矩形区域中绘制这个图像(另一个获取窗口部件的图像的方式是使用 `QPixmap::grabWidget()` 方法,把窗口部件作为参数传入)。

这种方法最主要的优点是它把几乎所有的绘制和样式方面的工作都交给了 Qt 本身去做,使得我们的代码比起用其他方式要极尽简单,而且得益于图像缓存的使用,我们的代码效率更高。

```
QSize RichTextDelegate::sizeHint(const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    QString html = index.model()->data(index, Qt::DisplayRole)
        .toString();
    document.setDefaultFont(option.font);
    document.setHtml(html);
    return QSize(document.idealWidth(), option.fontMetrics.height());
}
```

在大多数情况下,不需要重新实现 `QStyledItemDelegate::sizeHint()` 方法,但是这里遇到了非常情况。假设,例如有一段 HTML 文本“bold blue bear”,这段文本包含 54 个字母,但只有其中的 18 个会显示出来。标准的 `sizeHint()` 实现会想当然地基于全部的 54 个字符进行宽度的计算,所以必须重新实现这个方法以获得更准确的结果。

最明显的测定宽度的方法是把文本内容转换为纯文本并调用 `QFontMetrics::width()` 方法计算。遗憾的是,这种方法没有考虑更加精细的细节,例如作为上标或下标(通常字体总会更小些)的字符,或加粗、倾斜的字母通常要比普通字母要宽些,或者混合使用了多种不同的字体的情况。幸运的是,这里需要的精确计算(把我们提到的或更多的细节计入考量)已经由使用的 `QTextDocument::idealWidth()` 方法实现了。

在一些平台下,在每一次获取尺寸提示调用时都创建和销毁一个 `QTextDocument` 对象的代价非常昂贵,所以在类的私有成员中,声明了一个易变的 `mutableQTextDocument` 对象;也就是说,每次我们都重用同一个 `QTextDocument` 对象。

```
QWidget *RichTextDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &option, const QModelIndex&) const
{
    RichTextLineEdit *editor = new RichTextLineEdit(parent);
    editor->viewport()->setFixedHeight(option.rect.height());
    connect(editor, SIGNAL(returnPressed()),
        this, SLOT(closeAndCommitEditor()));
    return editor;
}
```

这个方法用来为指定模型索引代表的项创建一个合适的编辑器。因为这个委托用做包含富文本内容的列的列委托,我们不需要知道哪一个特定的项需要这个编辑器(稍后将介绍这个方法的另一个实现,返回的编辑器类型依赖于项的某些信息——例如,它是属于哪一列)。

我们创建了一个 `RichTextLineEdit` 部件(将在第9章介绍)来编辑委托将要处理的 HTML 数据。在本例中设置编辑器的可视高度为固定值(明确指出是参数 `option` 所指的矩形区域的高度)是必要的,用以防止 `RichTextLineEdit` 部件在文本输入时,其内容的忽上忽下的变化(这种情况的发生是因为 `RichTextLineEdit` 被设计为编辑单行内容,但它实际上是被设计为编辑多行内容的 `QTextEdit` 类的派生类)。

如果用户按下 `Return`(或 `Enter`) 按键,就把它看做是编辑完成的确认,所以我们把 `RichTextLineEdit` 的 `returnPressed()` 信号(模仿 `QLineEdit` 的同名信号)连接到一个私有自定义槽 `closeAndCommitEditor()` 上,我们一会将介绍。

```
void RichTextDelegate::setEditorData(QWidget *editor,
    const QModelIndex &index) const
{
    QString html = index.model()->data(index, Qt::DisplayRole)
        .toString();
    RichTextLineEdit *lineEdit = qobject_cast<RichTextLineEdit*>(
        editor);
    Q_ASSERT(lineEdit);
    lineEdit->setHtml(html);
}
```

一旦编辑器创建完毕,委托调用 `setEditorData()` 方法用从模型中获取的数据来初始化它。这里,我们先获取数据文本(HTML 格式的),然后获取通过 `createEditor()` 创建的 `RichTextLineEdit` 对象的指针,并设置编辑器的文本内容为前面获得的对应项的文本内容。

```
void RichTextDelegate::closeAndCommitEditor()
{
    RichTextLineEdit *lineEdit = qobject_cast<RichTextLineEdit*>(
        sender());
    Q_ASSERT(lineEdit);
    emit commitData(lineEdit);
    emit closeEditor(lineEdit);
}
```

实现一个像这样的槽对于需要一个信号来指示编辑过程已成功完成(比如 `QLineEdit` 的 `returnPressed()` 信号)的编辑器来说往往很有用。

我们使用 `qobject_cast < > ()` 连同 `QObject::sender()` 方法来获得一个指向 `RichTextLineEdit` 对象的指针,然后发射两个信号,一个信号通知委托提交编辑器的数据(即复制编辑器的数据到模型中),另一个信号通知委托关闭编辑器,因为不再需要编辑器了。

```
void RichTextDelegate::setModelData(QWidget *editor,
    QAbstractItemModel *model, const QModelIndex &index) const
{
    RichTextLineEdit *lineEdit = qobject_cast<RichTextLineEdit*>(
        editor);
    Q_ASSERT(lineEdit);
    model->setData(index, lineEdit->toSimpleHtml());
}
```

如果用户确认编辑完毕(通过在编辑器外部点击、通过制表符键切换到其他项,或通过按下 `Return` 或 `Enter` 按键),将调用上面的这个 `setModelData()` 方法(用户可以通过按下 `Esc` 键来取消编辑)。这里,先获取指向 `RichTextLineEdit` 对象的指针,然后把模型的文本内容设置为编辑器里的 HTML 内容。如我们将在第 9 章看到, `toSimpleHtml()` 方法比 `QTextEdit::toHtml()` 方法产生一个更加简单和紧凑的 HTML 内容,但有个限制即它只能处理一个非常有限的 HTML 子集。

在 `setEditorData()`、`closeAndCommitEditor()` 和 `setModelData()` 方法中用到了 `Q_ASSERT()` 来检查 `qobject_cast < > ()` 调用是否成功。我们倾向于当应用程序的逻辑在某个特定点指示某些内容必须为 `true` 的时候(例如,如果不为 `true`,这就是个 bug)使用断言,其他情况下使用条件语句(如 `if` 语句)(贯穿全书,我们能看到许多使用这两种方式的例子)。

到现在,我们就完成了 `RichTextDelegate` 的实现。除了 `paint()` 方法之外,所有方法的实现都很简单明了。这是由于该委托是用于数据类型相关(在本例中是富文本类型)的列(或行)委托,也就是说,它从来不需要去检查给定项的行或列来确认要处理的数据类型,而且总是用相同的方式处理每一项。

5.3 与模型相关的委托

如果我们不创建大量的模型,或许在需要时创建自定义的模型相关的委托,而不是创建一组更通用的列或行数据相关的委托会更方便些。在这一节将介绍一个模型相关的委托的典型示例,这个委托是在第 3 章我们见到过的 `Zipcodes` 应用程序中用到的,在图 3.3 所示的编辑一个州(state)的动作(action)时有所展示。

我们称这个委托为 `ItemDelegate`,并且将之定义为 `QStyledItemDelegate` 的子类。构造函数(没有有在文中介绍)把它的 `parent` 参数传递给了基类构造函数,函数体为空。在 `QStyledItemDelegate` API 之外还重新实现了 `paint()` 方法(仅用来绘制邮编,即 `zipcodes`,其他的内容都直接使用基类的实现),还有编辑数据的三个方法,即 `createEditor()`、`setEditorData()` 和 `setModelData()`。再不需要重新实现任何其他方法了,这是一个相当常见的情况。按照惯例,我们将依次介绍每一个方法。

```
void ItemDelegate::paint(QPainter *painter,
    const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    if (index.column() == Zipcode) {
        QStyleOptionViewItemV4 opt(option);
```

```

initStyleOption(&opt, index);
QString text = QString("%1").arg(opt.text.toInt(),
                                5, 10, QChar('0'));

painter->save();
painter->setRenderHints(QPainter::Antialiasing|
                      QPainter::TextAntialiasing);
if (opt.state & QStyle::State_Selected) {
    painter->fillRect(opt.rect, opt.palette.highlight());
    painter->setPen(opt.palette.highlightedText().color());
}
else
    painter->setPen(opt.palette.windowText().color());
painter->drawText(opt.rect.adjusted(0, 0, -3, 0), text,
                 QTextOption(Qt::AlignVCenter|Qt::AlignRight));
painter->restore();
}
else
    QStyledItemDelegate::paint(painter, option, index);
}

```

选择自己来绘制邮编 (zipcodes), 而把其他数据的绘制留给基类。注意到在这个方法以及在其他接下来就要介绍的方法中, 使用了 `index.column()` 来断定要处理的数据的列 (即数据类型), 对于在前一节中我们看到的列或行相关的委托是没有必要的, 因为它们被设置为应用于其相关模型中的特定行或列。

`QStyleOptionViewItem` 类是在 Qt 4.0 中引入的, 在随后的 Qt 4.x 系列版本中又相继增补了 `QStyleOptionViewItemV2`, `QStyleOptionViewItemV3` 和 `QStyleOptionViewItemV4`, 每个类都增加了新的公有成员。一般来说, 使用参数传入的 `QStyleOptionViewItem` 工作得很好, 但在一些情况下我们更愿意使用后续的版本, 这样就能利用那些额外的数据成员所带来的便利。

正确获取后续版本的 `QStyleOptionViewItem` 方法是依照下面介绍的模式: 创建一个 `QStyleOptionViewItemV4` (或者任何一个需要的版本) 对象并把参数 (`QStyleOptionViewItem` 类型) 传递给构造函数, 然后调用 `QStyleItemDelegate::initStyleOption()` 方法, 把新样式选项 (`QStyleOptionViewItemV4` 或其他版本类型对象) 和要处理的项的模型索引作为参数传递进去。

通过获取一个 `QStyleOptionViewItemV4` 对象, 就能够访问到它的 `text` 成员 (含有给定模型索引的项的文本内容), 而不需要使用 `index.model() -> data(index).toString()` 来获取 (尽管在这个示例中, 用到了 `text` 成员的 `toInt()` 方法)。

获取邮编后 (zipcode), 把它转化为整数, 然后创建一个字符串, 用我们想要的格式表示这个数字 (在本例中, 要求正好为 5 位数字, 如果有需要的话, 那么加前导 0)。

对于绘制过程, 先保存绘制对象的状态。在委托中如果想改变绘制对象的状态, 则保存绘制对象的状态是必要的, 因为考虑到效率, 视图的所有项都重复使用同一个绘制对象。我们启用了反走样 (antialiasing) 功能, 因为像先前的脚注中所说的, 我们不能确定默认值是什么, 所以谨慎的方式总是明确指定想要的渲染提示。如果该项是选中的就使用调色板的高亮色来绘制背景, 并相应地设置合适的画笔 (前景) 色。然后在给定的矩形区域中绘制文本内容, 右对齐并留有 3 个像素的右边距 (通过稍微收缩矩形区域获得), 这样文本内容就不会紧挨着表格单元的边框, 也就不会与单元格的边框线 (outline) 冲突了。在最后恢复绘制对象先前的状态, 为下一项的绘制做好准备。

```

QWidget *ItemDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    static QStringList usStates;
    if (usStates.isEmpty())
        usStates << "(Unknown)" << "Alabama" << "Alaska"

```

```

        ...
        << "West Virginia" << "Wisconsin" << "Wyoming";

    if (index.column() == Zipcode)
        return new ZipcodeSpinBox(parent);
    if (index.column() == State) {
        QComboBox *editor = new QComboBox(parent);
        editor->addItems(usStates);
        return editor;
    }
    return QStyledItemDelegate::createEditor(parent, option, index);
}

```

使用标准的 Qt 窗口部件来编辑大多数数据,但是使用了一个轻型的自定义微调框 (spinbox) 类来编辑邮编 (zipcodes)。createEditor() 方法必须判断应该使用哪一个编辑器 (在这个模型中,它取决于哪一列),然后创建、设置并返回编辑器,从被编辑项的数据中准备好要在编辑器中填充的数据,然后显示给用户。需要注意的是,如果我们自己创建了一个编辑器,那么有必要传递一个合适的 parent 参数给它——这将确保 Qt 获得该编辑器的所有权,并能在合适的时间删除它。

对于代表州 (state) 的列,使用一个组合下拉框 (combobox),并用“(Unknown)”和所有的美国州名来填充。把州名保存在一个静态的 QStringList 对象中,这样这些数据只会创建一次。

至于邮局 (post office) 和县 (county) 列 (使用纯文本),我们把工作交给基类去处理,将返回一个 QLineEdit——除非使用先前介绍过的 QItemEditorFactory 类设置了另一个不同的编辑器。

```

void ItemDelegate::setEditorData(QWidget *editor,
    const QModelIndex &index) const
{
    if (index.column() == Zipcode) {
        int value = index.model()->data(index).toInt();
        ZipcodeSpinBox *spinBox =
            qobject_cast<ZipcodeSpinBox*>(editor);
        Q_ASSERT(spinBox);
        spinBox->setValue(value);
    }
    else if (index.column() == State) {
        QString state = index.model()->data(index).toString();
        QComboBox *comboBox = qobject_cast<QComboBox*>(editor);
        Q_ASSERT(comboBox);
        comboBox->setCurrentIndex(comboBox->findText(state));
    }
    else
        QStyledItemDelegate::setEditorData(editor, index);
}

```

一旦编辑器创建完毕,Qt 的模型/视图架构会调用 setEditorData() 方法,在编辑器显示给用户之前会给我们一个填充编辑器数据的机会。我们必须总是处理那些自己创建的窗口部件,而把那些基类负责的窗口部件的工作交由基类去做。

这里用到的逻辑几乎总是相同的:获取给定模型索引对应的项,把 QWidget 类型的指针转化为一个正确类型的编辑器部件的指针,然后填充编辑器的数据。对于组合下拉框部件,我们用另一种稍微不同的方式实现,因为它已经包含了所有有效数据,所以只要把它的当前文本内容 (current text) 设置为模型中对应项的文本内容即可。

```

void ItemDelegate::setModelData(QWidget *editor,
    QAbstractItemModel *model, const QModelIndex &index) const
{
    if (index.column() == Zipcode) {
        ZipcodeSpinBox *spinBox =
            qobject_cast<ZipcodeSpinBox*>(editor);

```

```

        Q_ASSERT(spinBox);
        spinBox->interpretText();
        model->setData(index, spinBox->value());
    }
    else if (index.column() == State) {
        QComboBox *comboBox = qobject_cast<QComboBox*>(editor);
        Q_ASSERT(comboBox);
        model->setData(index, comboBox->currentText());
    }
    else
        QStyledItemDelegate::setModelData(editor, model, index);
}

```

如果用户确认编辑完毕,对于我们负责的那些编辑器部件,必须获取编辑器的值,并把它设置为给定模型索引所代表的项的值。对于其他窗口部件,把工作交由基类去做即可。

对于微调框(spinbox),我们使用了一种十分谨慎的方式,调用 `interpretText()` 方法来确认,如果用户通过输入或删除数字改变了值,而不是通过使用微调按钮,微调框中保存的值应该正确地反映用户的修改。对于组合下拉框,则使用更加简便的方式,仅获取当前值即可。

为了完整起见,这里给出了 `ZipcodeSpinBox` 类的全部定义:

```

class ZipcodeSpinBox : public QSpinBox
{
    Q_OBJECT
public:
    explicit ZipcodeSpinBox(QWidget *parent)
        : QSpinBox(parent)
    {
        setRange(MinZipcode, MaxZipcode);
        setAlignment(Qt::AlignVCenter|Qt::AlignRight);
    }
protected:
    QString textFromValue(int value) const
    { return QString("%1").arg(value, 5, 10, QChar('0')); }
};

```

如果只需要设置微调框的范围和文本对齐方式,只要使用一个标准的 `QSpinBox` 并在 `createEditor()` 方法中做相关设置即可。我们选择在构造函数中进行设置,因为不管怎样都需要从 `QSpinBox` 派生,这样才能重写 `textFromValue()` 方法。这样做以使微调框的文本内容能正确的以五位数字(如有必要的话辅以前导的0,)来呈现邮编(zipcode)值——绘制时将以同样的格式来显示邮编值。

现在我们完成了一个典型的模型相关的自定义委托的介绍。这样的委托不像那些一般可用于任何模型的数据类型相关的行或列委托那样通用,但它们把所有的委托代码保存在一个地方,给了我们完全的、直接控制模型中的项的外观和编辑方式的能力。如我们在这里所看到的,可以时常把一些工作交给基类去做。

使用自定义委托是最通用和最便捷的控制模型项的呈现和编辑的方式。然而,如果想以一种不同于任何一种 Qt 内置视图的方式呈现数据项,或者想自定义互相关联的项的外观(例如以某种方式联合显示一项或多项),这时就需要创建自定义视图了。创建自定义视图是下一章的主题。

第6章 模型/视图中的视图

- QAbstractItemView 子类
- 与模型相关的可视化视图

本章介绍模型/视图中的视图,并且是本书中介绍 Qt 的模型/视图架构内容的最后一章。就像前两章一样,这里假定用户如第3章开始时所说的那样对模型/视图架构已基本熟悉。

大多数时候,Qt 中的标准模型视图,即 QListView、QTableView、QColumnView 和 QTreeView 对大多数场合来说足够用了。像其他 Qt 类一样,它们都能被子类化或者能使用自定义委托,以影响模型项的显示。然而,有两种需要创建自定义视图的情况,第一种是我们想呈现数据的方式与 Qt 的标准视图呈现数据的方式从根本上不一样,另一种是我们想以某种方式把两项或多项的数据项组合起来进行显示。

广义地讲,有两种创建自定义视图的方式可采用。一种方式在我们想要创建一个视图组件时使用,即一个可能在多个不同的模型间进行复用,并且与 Qt 的模型/视图架构相适应的视图。在这些情况下我们通常子类化 QAbstractItemView,并且提供标准的视图 API,这样任何模型都能使用我们的视图。另一种方式,当我们想以一种独特的方式显示一个特定模型中的数据,并且这种显示方式没什么可能或根本没有可能进行复用时比较有用。在这些情况下,我们只能简单地创建一个刚好有且仅有所需功能的自定义模型视图。这通常涉及到子类化 QWidget 并提供自己的 API,但包含了一个 setModel() 方法。

在这一章中将介绍两个关于自定义视图的示例。第一个是一般的 QAbstractItemView 子类,提供了与 Qt 内置视图相同的 API,可以和任何模型一起使用,尽管它被特别设计为呈现和编辑列表模型的。第二个是专用于一个特定模型的可视化视图,它提供了自己的 API。

6.1 QAbstractItemView 子类

在本节将展示如何创建一个可用来替换 Qt 的标准视图的 QAbstractItemView 子类。在实践中,就像列表、表格和树模型那样,它们当然有对应的视图,所以这里我们将开发一个自定义列表视图,尽管在原理上是与所有的 QAbstractItemView 子类相同的。

图 6.1 所示的是平铺列表视图(Tiled List View)应用程序(tiledlistview)的中心区域。这部分有两个使用同一个模型的视图:左边是一个标准的 QListView,右边是一个 TiledListView。我们注意到尽管两个窗口部件大小相同并且使用相同的字体,TiledListView 却显示更多的数据。另外,如图中所示,TiledListView

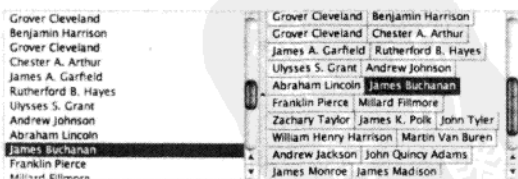


图 6.1 一个 QListView 和一个 TiledListView

没有使用多个列,而是在每行显示尽可能多的项。例如,如果我们把尺寸加宽些,在一些行中就有可能放下四个或更多的项。

TiledListView 在易用性方面的一个不同点,使键盘导航(navigation)变得更快、更方便,也更合理了,那就是使用方向键不仅仅只是在项的列表中前后移动,当用户使用向上键(或向下键)在各

个项之间移动的时候,选中项变为可见的当前项上方(或下方)的项。与此相似,当用户使用向左(或向右)键浏览时,选中项中如期望的那样变为当前项左边(或右边)的项,除非当前项已经是左边(或最右边)的项了。对于靠近边缘的情形,选中项切换到在逻辑上当前项之前(或之后)的那一项。

`QAbstractItemView` 的 API 比较多,在写作本章之时,Qt 文档没有明确地说明哪些 API 是必须通过子类化重新实现的,以及基类中的哪些实现是足够使用的。然而,有些方法是纯虚(pure virtual)函数所以必须要重新实现。另外,Qt 同时还提供了 `examples/itemviews/chart` 示例作为一个自定义视图实现的帮助指导。

在表 6.1 中列出了在 `TiledListView` 中已经实现了的那些 API,以及我们认为实现一个自定义 `QAbstractItemView` 派生类所需的最小 API 集合。Qt 中的 `chart` 示例重新实现了表格中所列的所有方法,还有 `mouseReleaseEvent()` 和 `mouseMoveEvent()` 事件处理函数[以提供橡皮筋(rubber band)的支持,在 `TiledListView` 中是不需要的]。`chart` 示例还实现了 `edit()` 方法以初始化编辑过程,尽管 `TiledListView` 是可编辑的,但这个方法在 `TiledListView` 中也不需要重新实现,因为从基类中继承过来的行为足够用了。

表 6.1 `QAbstractItemView` 的 API

方 法	说 明
<code>dataChanged(topLeft, bottomRight)</code>	当模型中由 <code>topLeft</code> 到 <code>bottomRight</code> 涵盖的模型索引对应的项发生变化时调用这个槽
<code>horizontalOffset()</code> [†]	返回视图的水平偏移量
<code>indexAt(point)</code> [†]	返回视口(viewport)中坐标点 <code>point</code> 处的项的模型索引
<code>isIndexHidden(index)</code> [†]	如果模型索引 <code>index</code> 所对应的项是隐藏状态(所以应该不显示的),则返回 <code>true</code>
<code>mousePressEvent(event)</code>	通常用于把鼠标点击的项的模型索引设置为当前模型索引
<code>moveCursor(how, modifiers)</code> [†]	在按照 <code>how</code> (例如向上、向下、向左或向右)的指示进行移动后返回新项的模型索引,同时把由 <code>modifiers</code> 指明的控制按键计入考量
<code>paintEvent(event)</code>	绘制视图的内容到视口
<code>resizeEvent(event)</code>	通常用于更新滚动条
<code>rowsAboutToBeRemoved(parent, start, end)</code>	当以 <code>parent</code> 为父索引的从 <code>start</code> 开始到 <code>end</code> 结束的行要被删除的时候调用该方法
<code>rowsInserted(parent, start, end)</code>	当从 <code>start</code> 开始到 <code>end</code> 结束的行已被插入到 <code>parent</code> 父索引之下时调用该方法
<code>scrollContentsBy(dx, dy)</code>	在水平方向和垂直方向上分别滚动视图的视口 <code>dx</code> 和 <code>dy</code> 像素
<code>scrollTo(index, hint)</code> [†]	滚动视图以确保给定模型索引 <code>index</code> 所对应的项是可见的,按照参数 <code>hint</code> 所指的滚动属性来滚动
<code>setModel(model)</code>	使视图使用给定的模型 <code>model</code>
<code>setSelection(rect, flags)</code>	应用选中标志 <code>flags</code> 到参数 <code>rect</code> 涵盖或接触到的所有项
<code>updateGeometries()</code>	通常用于更新视图的子窗口部件的位置和尺寸,例如滚动条
<code>verticalOffset()</code>	返回视图的垂直偏移量
<code>visualRect(index)</code> [†]	返回给定模型索引 <code>index</code> 对应的项所占据的矩形区域
<code>visualRegionForSelection(selection)</code> [†]	返回参数 <code>selection</code> 中包含的那些项的视口区域

[†] 这是个纯虚方法,因此必须在派生类中重新实现。

在介绍 `TiledListView` 类之前,我们先看看它的实例是如何创建和初始化的。

```
TiledListView *tiledListView = new TiledListView;
tiledListView->setModel(model);
```

这两行代码表现得很清楚,`TiledListView` 在使用方式上与其他视图类完全相同。

既然表 6.1 中已列出来那些必须实现的 API,我们就不显示头文件中相关的定义了,除了私有数据成员(所有的私有数据成员都是 `TiledListView` 专有的)。

```
private:
    mutable int idealWidth;
    mutable int idealHeight;
    mutable QHash<int, QRectF> rectForRow;
    mutable bool hashIsDirty;
```

成员变量 `idealWidth` 和 `idealHeight` 是在不需要出现滚动条的情况下显示所有项所需要的宽度和高度。`rectForRow` 哈希表返回包含指定行的正确位置信息和大小的 `QRectF` (要注意的是, 因为 `TiledListView` 是设计为显示列表的, 所以一行对应的是一项)。所有的这些成员变量都和后台状态记录有关, 另外由于它们都是在 `const` 方法中使用的, 我们强制它们都加了 `mutable` 关键字。

我们不在一有变化发生时就更新 `rectForRow` 哈希表, 而只做“懒惰更新”(lazy update)——即在变化发生时仅仅设置 `hashIsDirty` 成员变量为 `true`, 然后, 当什么时候需要访问 `rectForRow` 哈希表, 且 `hashIsDirty` 为 `true` 时, 才会去重新计算。

现在我们差不多准备好了来介绍 `TiledListView` 的实现了, 现在就开始, 先从构造函数讲起, 还包括那些必要的私有方法。但我们首先必须提到一个关于 `QAbstractItemView` 派生类的重要概念。

`QAbstractItemView` 基类为它要显示的内容提供了一个滚动区域。一个 `QAbstractItemView` 子类窗口部件的唯一可见区域是它的视口 (viewport), 就是在滚动区域中显示出来的那部分。这个可见区域可以通过 `viewport()` 方法访问到。这个窗口部件的实际大小无关紧要; 重要的是要显示所有的模型数据所需要的窗口部件尺寸 (即使它远远超出屏幕大小)。在介绍 `calculateRectsIfNecessary()` 和 `updateGeometries()` 方法时将会看到这一点是如何影响我们的代码的。

```
TiledListView::TiledListView(QWidget *parent)
    : QAbstractItemView(parent), idealWidth(0), idealHeight(0),
      hashIsDirty(false)
{
    setFocusPolicy(Qt::WheelFocus);
    setFont(QApplication::font("QListView"));
    horizontalScrollBar()->setRange(0, 0);
    verticalScrollBar()->setRange(0, 0);
}
```

构造函数调用了基类构造函数并且初始化了一些私有数据成员。初始化视图的“理想”尺寸为 `0×0`, 因为现在还没有数据要显示。

不同于以往, 我们调用 `setFont()` 方法来设置窗口部件的字体, 而不是像通常在自定义窗口部件中所做的那样使用继承过来的字体。字体由 `QApplication::font()` 方法返回, 如果给定一个类名, 就会返回一个该类使用的平台相关的字体。这将使 `TiledListView` 使用正确的字体, 尽管那些平台 (例如 Mac OS X) 可能会为 `QListView` 部件使用与默认的 `QWidget` 部件字体稍微不同尺寸的字体^①。

因为现在还没有数据, 所以我们设置滚动条的范围为 `(0, 0)`; 这将确保滚动条一直隐藏直到需要时才显示, 而把关于隐藏和显示方面的工作留给基类去处理。

```
void TiledListView::setModel(QAbstractItemModel *model)
{
    QAbstractItemView::setModel(model);
    hashIsDirty = true;
}
```

当设置一个模型的时候, 首先调用基类的实现, 然后设置私有数据成员 `hashIsDirty` 标志为 `true`, 以确保调用 `calculateRectsIfNecessary()` 方法时, 它将更新 `rectForRow` 哈希表。

① 关于 Qt 的字体和调色板如何工作这方面的更多内容, 请参阅 labs.qt.nokia.com/blogs/2008/11/16。

`indexAt()`、`setSelection()`和`viewportRectForRow()`方法都需要知道模型中的项的尺寸和位置。对于`mousePressEvent()`、`moveCursor()`、`paintEvent()`和`visualRect()`方法也间接地需要这些信息,因为这4个方法都调用到了需要用到项的尺寸和位置的方法。我们不在每次需要的时候都动态计算这些矩形区域(尺寸和位置),而是选择通过把它们缓存在`rectForRow`哈希表中,用一些内存换取速度。另外我们不是在每次变化发生时都调用`calculateRectsIfNecessary()`方法来保持哈希表中的数据是最新的,而是仅仅记录哈希表是否已经过时了,而只在真正需要访问哈希表的时候才重新计算这些矩形区域。

```
const int ExtraHeight = 3;

void TiledListView::calculateRectsIfNecessary() const
{
    if (!hashIsDirty)
        return;
    const int ExtraWidth = 10;
    QFontMetrics fm(font());
    const int RowHeight = fm.height() + ExtraHeight;
    const int MaxWidth = viewport()->width();
    int minimumWidth = 0;
    int x = 0;
    int y = 0;
    for (int row = 0; row < model()->rowCount(rootIndex()); ++row) {
        QModelIndex index = model()->index(row, 0, rootIndex());
        QString text = model()->data(index).toString();
        int textWidth = fm.width(text);
        if (!(x == 0 || x + textWidth + ExtraWidth < MaxWidth)) {
            y += RowHeight;
            x = 0;
        }
        else if (x != 0)
            x += ExtraWidth;
        rectForRow[row] = QRectF(x, y, textWidth + ExtraWidth,
                                RowHeight);
        if (textWidth > minimumWidth)
            minimumWidth = textWidth;
        x += textWidth;
    }
    idealWidth = minimumWidth + ExtraWidth;
    idealHeight = y + RowHeight;
    hashIsDirty = false;
    viewport()->update();
}
```

这个方法是 `TiledListView` 的核心方法,至少就外观而言,就如一会就要看到的,所有的绘制都要用到在这个方法中创建的矩形区域。

首先看看那些矩形区域是否需要重新计算。如果需要重新计算,就先计算出显示一行所需要的高度,以及在视口中可用的最大宽度,即可用的可见宽度。

在这个方法的主循环中,遍历模型中的每一行(即每一项),然后获取项的文本内容。接着计算该项所需要的宽度以及该项应该显示的 x 坐标和 y 坐标——这取决于该项能否在与前一项所在的同一行(即在同一可见行)中合适地显示出来,或者是否必须开始一个新行。一旦知道了该项的尺寸和位置,就由此(尺寸、位置)创建一个矩形区域,并把它以该项的行号作为键值加入到 `rectForRow` 哈希表中。

需要注意的是,在循环中进行计算的过程中,使用的是实际的可见宽度,但是假设了可用高度是在这个宽度下能够显示所有项所需要的高度。为了获得想要的模型索引,把 `QAbstractItemView::rootIndex()` 而不是一个无效的模型索引(`QModelIndex()`)作为父索引参数。这两者都在列

表模型中工作得一样出色,但在 `QAbstractItemView` 的派生类中使用更加通用的 `rootIndex()` 是一种更好的风格。

在末尾重新计算了理想的宽度值(即最宽的项的宽度加上一些边距)和理想的高度值(高度是在视口的当前宽度下能够显示所有项所必需的高度,而不管视口的真实高度是多少)——这时候 y 变量保存的是所有行的总共高度。理想宽度可能比可用宽度要大些,例如,如果视口宽度比要显示的最宽项的宽度要窄的话(在这种情况下,水平滚动条将自动显现出来)。一旦计算完毕,我们就在视口上调用 `update()` 方法(因为所有的重绘工作都只在视口上进行,而不是在 `QAbstractItemView` 自定义窗口部件自身上进行),这样数据就会被重绘。

去关注 `QAbstractItemView` 自定义窗口部件本身的真实尺寸是没意义的——所有的计算工作都是依照视口以及理想宽度和高度进行的。

```
QRect TiledListView::visualRect(const QModelIndex &index) const
{
    QRect rect;
    if (index.isValid())
        rect = viewportRectForRow(index.row()).toRect();
    return rect;
}
```

这个纯虚方法必须返回给定模型索引所对应的项占据的矩形区域,它的实现非常简单,因为我们把工作交给了从 `rectForRow` 哈希表中获取数据的私有方法 `viewportRectForRow()` 来完成。

```
QRectF TiledListView::viewportRectForRow(int row) const
{
    calculateRectsIfNecessary();
    QRectF rect = rectForRow.value(row).toRect();
    if (!rect.isValid())
        return rect;
    return QRectF(rect.x() - horizontalScrollBar()->value(),
                  rect.y() - verticalScrollBar()->value(),
                  rect.width(), rect.height());
}
```

`visualRect()`、`moveCursor()` 和 `paintEvent()` 方法用到上面的这个方法。它返回某一项的最大精度的 `QRectF` 对象(例如对于 `paintEvent()` 方法);其他调用者则使用 `QRectF::toRect()` 方法把返回值转化为了普通的基于整型的 `QRect` 对象。

访问 `rectForRow` 哈希表的方法在访问发生前必须调用 `calculateRectsIfNecessary()` 方法。如果 `rectForRow` 哈希表是最新的,那么 `calculateRectsIfNecessary()` 方法什么也不做;否则它将重新计算哈希表中的矩形区域以备使用。

`rectForRow` 哈希表中的矩形区域有每一行(即项)的 x 和 y 坐标,这些坐标基于理想宽度(通常是可视宽度)和理想高度(在当前宽度下显示所有项所需要的高度)。这就是说,这些矩形区域有效地使用了基于该窗口部件的理想尺寸(窗口部件的实际大小是不相关的)的窗口部件坐标系统。`viewportRectForRow()` 方法必须返回一个在视口坐标系统中的矩形区域,这样我们才能调整坐标把滚动(scrolling)计入考量。图 6.2 演示了窗口部件坐标系统和视口坐标系统之间的不同之处。

```
bool isIndexHidden(const QModelIndex&) const { return false; }
```

必须重新实现上面这个纯虚方法,因为它的实现很简单,所以放到了头文件中。这个方法是为那些可能有隐藏项的数据设计的。例如,一个有隐藏的行或列的表格。对于这个视图来说,没有任何项是隐藏的,因为我们不支持隐藏项,所以这里总是返回 `false`。

```

void TiledListView::scrollTo(const QModelIndex &index,
                           QAbstractItemView::ScrollHint)
{
    QRect viewRect = viewport()->rect();
    QRect itemRect = visualRect(index);

    if (itemRect.left() < viewRect.left())
        horizontalScrollBar()->setValue(horizontalScrollBar()->value()
        + itemRect.left() - viewRect.left());
    else if (itemRect.right() > viewRect.right())
        horizontalScrollBar()->setValue(horizontalScrollBar()->value()
        + qMin(itemRect.right() - viewRect.right(),
        itemRect.left() - viewRect.left()));
    if (itemRect.top() < viewRect.top())
        verticalScrollBar()->setValue(verticalScrollBar()->value() +
        itemRect.top() - viewRect.top());
    else if (itemRect.bottom() > viewRect.bottom())
        verticalScrollBar()->setValue(verticalScrollBar()->value() +
        qMin(itemRect.bottom() - viewRect.bottom(),
        itemRect.top() - viewRect.top()));
    viewport()->update();
}

```

这是另一个不得不实现的纯虚方法。幸运的是,它的实现是简明易懂的(几乎与 Qt 的 chart 示例中的几乎一模一样)。

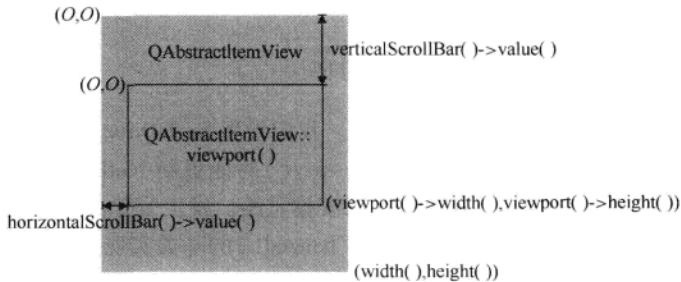


图 6.2 窗口部件与视图坐标系统对比

如果指定项占据的矩形区域在视图的左边框的左边,那么视图必须要滚动。滚动是通过改变水平滚动条的值来实现的,即加上指定项占据的矩形区域的左边框与视图的左边框距离之差。其他情况的实现与此类似。

需要注意的是,这个方法调用 `visualRect()` 方法,`visualRect()` 又调用 `viewportRectForRow()` 方法,而 `viewportRectForRow()` 方法又调用了 `calculateRectsIfNecessary()` 方法——如同先前已经提醒注意的,如果哈希表有所改变的话,这最后一个方法会重新计算 `rectForRow` 哈希表中的矩形区域。

```

QModelIndex TiledListView::indexAt(const QPoint &point_) const
{
    QPoint point(point_);
    point.rx() += horizontalScrollBar()->value();
    point.ry() += verticalScrollBar()->value();
    calculateRectsIfNecessary();
    QHashIterator<int, QRectF> i(rectForRow);
    while (i.hasNext()) {
        i.next();
        if (i.value().contains(point))
            return model()->index(i.key(), 0, rootIndex());
    }
    return QModelIndex();
}

```

这个纯虚方法必须返回给定坐标点对应的项的模型索引。这个坐标点是基于视口坐标系,但是 `rectForRow` 中的那些矩形区域是基于窗口部件的坐标系。我们不是把每一个矩形区域坐标进行转换来检查它是否包含这个坐标点,而是只做一次转换,把该坐标点的坐标转换为窗口部件坐标系的坐标。

`QPoint::rx()` 和 `QPoint::ry()` 方法返回该坐标点的 x 坐标和 y 坐标的一个非常量引用,这样就易于改变它们了。如果没有这样的方法,就不得不做像 `point.setX(horizontalScrollBar() -> value() + point.x())`。

首先要确保 `rectForRow` 哈希表是最新的,然后遍历哈希表中的每一行(也就是项)——以任意顺序,因为哈希表是没有排序的。如果找到了对应值,即找到了一个矩形区域包含了该点,那么立即返回它对应的模型索引。

对于含有大量的项(超过了数千)的模型,这种方式可能运行得比较慢,因为在最坏的情况下,每一项的矩形区域都需要检查,即使平均起来,也要检查一半的项。对于 `TiledListView` 来说这不大可能是个问题,因为把数千个项加入到一个任何类型的列表模型中对用户来说可能没什么用处——使用一个把项进行了分组、并使顶级项设置为更易于管理的数目的树模型应该是个更好的方法。

```
void TiledListView::dataChanged(const QModelIndex &topLeft,
                               const QModelIndex &bottomRight)
{
    hashIsDirty = true;
    QAbstractItemView::dataChanged(topLeft, bottomRight);
}
```

当模型数据发生改变时,上面这个方法将被调用。我们设置 `hashIsDirty` 为 `true` 以确保在下次将要访问哈希表数据的时候,调用 `calculateRectsIfNecessary()` 时能更新 `rectForRow`,然后调用基类的实现。注意到这里没有调用 `viewport -> update()` 来安排重绘。发生变化的数据可能是不可见的,所以重绘可能不是必要的,如果确实需要重绘,`dataChanged()` 的基类实现会代我们做重绘调度。

```
void TiledListView::rowsInserted(const QModelIndex &parent, int start,
                                 int end)
{
    hashIsDirty = true;
    QAbstractItemView::rowsInserted(parent, start, end);
}

void TiledListView::rowsAboutToBeRemoved(const QModelIndex &parent,
                                          int start, int end)
{
    hashIsDirty = true;
    QAbstractItemView::rowsAboutToBeRemoved(parent, start, end);
}
```

如果有新行插入到了模型中,或者有一些行将要被删除,那么必须确保视图能正确响应。这种情况只需要简单地把工作交由基类完成就可以了;我们必须做的是,标记 `rectForRow` 哈希表为已发生变化(`dirty`)的,这样它将在需要的时候重新进行计算——例如,如果基类方法安排了重绘时。

```
QModelIndex TiledListView::moveCursor(
    QAbstractItemView::CursorAction cursorAction,
    Qt::KeyboardModifiers)
{
    QModelIndex index = currentIndex();
    if (index.isValid()) {
        if ((cursorAction == MoveLeft && index.row() > 0) ||
            (cursorAction == MoveRight &&
             index.row() + 1 < model()->rowCount())) {
```

```

const int offset = (cursorAction == MoveLeft ? -1 : 1);
index = model()->index(index.row() + offset,
                      index.column(), index.parent());
}
else if ((cursorAction == MoveUp && index.row() > 0) ||
        (cursorAction == MoveDown &&
         index.row() + 1 < model()->rowCount())) {
    QFontMetrics fm(font());
    const int RowHeight = (fm.height() + ExtraHeight) *
                          (cursorAction == MoveUp ? -1 : 1);
    QRect rect = viewportRectForRow(index.row()).toRect();
    QPoint point(rect.center().x(),
                 rect.center().y() + RowHeight);
    while (point.x() >= 0) {
        index = indexAt(point);
        if (index.isValid())
            break;
        point.rx() -= fm.width("n");
    }
}
return index;
}

```

就像 `calculateRectsIfNecessary()` 方法是 `TiledListView` 的外观的核心,上面这个方法是它的行为的核心。这个方法必须返回所请求移动动作要移动到的位置处的项的模型索引——如果没有移动动作发生,就返回一个无效的模型索引。

如果用户按了向左方向键(或向右方向键)我们必须返回列表中当前项的前一项(或后一项)的模型索引——或者返回当前项,如果前一项(或后一项)已经是列表模型的第一项(或最后一项)的话。这很简单地通过基于当前项的模型索引项,但用前一行(或后一行)的行号创建一个新的模型索引来获得。

处理向上方向键和向下方向键要比处理向左方向键和向右方向键稍微麻烦一些。在两种情况下,必须计算出当前项的上方或下方的一个坐标点。计算出来的坐标点是否处于视口之外不重要,只要它是处于某一项的矩形区域之内就行。

如果用户按下了向上方向键(或向下方向键)我们必须返回显示在当前项上方(或下方)的那一项的模型索引。首先获取到当前项在视口中的矩形区域,接着创建一个恰好在当前项垂直方向上方(或下方)一行、水平方向在项的中心处的点。然后使用 `indexAt()` 方法来获取前面计算出来的坐标点处的项的模型索引。如果获取到一个有效的模型索引,那么在当前项上方(或下方)确实有一项存在,而且我们有它的模型索引,这样就完成了工作,可以返回这个模型索引了。

但是这个模型索引可能是无效的:这是可能的,因为可能在当前项的上方(或下方)没有任何项。回头看看那个截图,在右边框附近的那些项参差不齐,因为每一行都有不同的长度。如果碰到这种情况,我们向左移动一个“n”字符的宽度然后再进行尝试,反复向左移动,直到找到一项(也就是直到获得一个有效的模型索引)或直到移动到超出了左边框,这就是说在它的上方(或下方)没有任何项。当前项处于第一行(或最后一行)时,用户按向上方向键(或向下方向键)是不会有项在它的上方(或下方)的。

如果 `moveCursor()` 方法返回一个无效的 `QModelIndex`, `QAbstractItemView` 基类什么也不做。

我们还没有写任何关于处理选集(selection)的代码——不需要去做,因为使用了 `QAbstractItemView` API。如果用户按下移动键的同时也按下了 `Shift` 键,将会创建一个包含连续项的选集。类似地,当用户同时按下 `Ctrl` 键(在 Mac OS X 上是 `⌘` 键),那么可以点击任意的项,每个项都会被选中并包含到一个选集中(可能包含不连续的项)。

我们把对于 Home、End、PageUp 和 PageDown 键的支持的实现作为一个练习留给读者——它们仅仅需要把 moveCursor() 方法扩展到支持更多的 CursorAction 类型(例如 QAbstractItemView::MoveHome 和 QAbstractItemView::MovePageUp)。

```
int TiledListView::horizontalOffset() const
{
    return horizontalScrollBar()->value();
}

int TiledListView::verticalOffset() const
{
    return verticalScrollBar()->value();
}
```

这两个纯虚方法必须重新实现。它们必须返回视口在窗口部件(理想尺寸)中的 x 坐标和 y 坐标偏移量(offset)。它们很容易实现,因为滚动条的值就是需要的偏移量。

```
void TiledListView::scrollContentsBy(int dx, int dy)
{
    scrollDirtyRegion(dx, dy);
    viewport()->scroll(dx, dy);
}
```

当滚动条移动时,就调用这个方法;它的职责是确保视口按给定的数值进行滚动,并安排适当的重绘。这里通过在进行滚动之前调用 QAbstractItemView::scrollDirtyRegion() 方法来设置重绘。除了调用 scrollDirtyRegion(), 另一种选择是在执行完滚动之后调用 viewport -> update() 方法。

基类的实现仅是调用了 viewport -> update(), 并没有真正执行滚动。需要注意的是, 如果想在程序中用编码方式执行滚动,就需要通过调用 QScrollBar::setValue() 作用于滚动条来实现,而不是调用上面这个 scrollContentsBy() 方法。

```
void TiledListView::setSelection(const QRect &rect,
                                QFlags<QItemSelectionModel::SelectionFlag> flags)
{
    QRect rectangle = rect.translated(horizontalScrollBar()->value(),
                                       verticalScrollBar()->value()).normalized();
    calculateRectsIfNecessary();
    QHashIterator<int, QRectF> i(rectForRow);
    int firstRow = model()->rowCount();
    int lastRow = -1;
    while (i.hasNext()) {
        i.next();
        if (i.value().intersects(rectangle)) {
            firstRow = firstRow < i.key() ? firstRow : i.key();
            lastRow = lastRow > i.key() ? lastRow : i.key();
        }
    }
    if (firstRow != model()->rowCount() && lastRow != -1) {
        QItemSelection selection(
            model()->index(firstRow, 0, rootIndex()),
            model()->index(lastRow, 0, rootIndex()));
        selectionModel()->select(selection, flags);
    }
    else {
        QModelIndex invalid;
        QItemSelection selection(invalid, invalid);
        selectionModel()->select(selection, flags);
    }
}
```

这个纯虚方法用于把给定的选择标志应用到给定矩形区域涵盖的或接触到的项中。真正的选择动作必须通过调用 QAbstractItemView::selectionModel() -> select() 来完成。这里的实现与 Qt 的 chart 示例中使用到的非常相似。

传入的矩形区域是基于视口的坐标系统的,所以首先创建一个基于窗口部件坐标系统的矩形区域,因为它将被 `rectForRow` 哈希表所使用。然后遍历哈希表中的所有行(也就是项)以任意顺序,如果任何一项所处的矩形区域与给定的矩形区域有交集,就扩展选集涵盖的起始行和结束行以包含该项(如果该项还没包含进来的话)。

如果选集的起始行和结束行有效,那么创建一个 `QItemSelection` 对象涵盖这些行(包括起始行和结束行)并且更新视图的选集模型(selection model)。但如果起始行和结束行之中有一个或两个都是无效的,那么创建一个无效的 `QModelIndex` 对象并用它来更新选集模型。

```
QRegion TiledListView::visualRegionForSelection(
    const QItemSelection &selection) const
{
    QRegion region;
    foreach (const QItemSelectionRange &range, selection) {
        for (int row = range.top(); row <= range.bottom(); ++row) {
            for (int column = range.left(); column < range.right();
                ++column) {
                QModelIndex index = model()->index(row, column,
                                                    rootIndex());
                region += visualRect(index);
            }
        }
    }
    return region;
}
```

这个纯虚方法必须重新实现,并且要返回一个 `QRegion` 对象,这个对象包含了视图中所有在视口中显示出来的选中项,并使用视口的坐标系统。上面使用的实现与 Qt 的 `chart` 示例中使用到的非常相似。

先创建一个空的 `region` 对象,然后遍历所有的选集(如果有的话)。对于每一个选集获取其中的每一项的模型索引,并把每一项的可视区域合并到 `region` 对象中。

`visualRect()` 方法的实现调用了 `viewportRectForRow()` 方法,从 `rectForRow` 哈希表中获取对应项的矩形区域,并把它转换为基于视口坐标系统(因为 `rectForRow` 里的矩形区域是基于窗口部件坐标系统的)的形式并返回。在这个特定的例子中可以绕过 `visualRect()` 调用而直接使用 `rectForRow` 哈希表,但我们宁可做更一般化的实现以便易于适应其他的自定义视图。

```
void TiledListView::paintEvent(QPaintEvent*)
{
    QPainter painter(viewport());
    painter.setRenderHints(QPainter::Antialiasing|
                          QPainter::TextAntialiasing);
    for (int row = 0; row < model()->rowCount(rootIndex()); ++row) {
        QModelIndex index = model()->index(row, 0, rootIndex());
        QRectF rect = viewportRectForRow(row);
        if (!rect.isValid() || rect.bottom() < 0 ||
            rect.y() > viewport()->height())
            continue;
        QStyleOptionViewItem option = viewOptions();
        option.rect = rect.toRect();
        if (selectionModel()->isSelected(index))
            option.state |= QStyle::State_Selected;
        if (currentIndex() == index)
            option.state |= QStyle::State_HasFocus;
        itemDelegate()->paint(&painter, option, index);
        paintOutline(&painter, rect);
    }
}
```



绘制视图出乎意料地简单明了,因为每一项占据的矩形区域早已计算出来了,可以从 `rectForRow` 哈希表中直接使用。但要注意我们是在窗口部件的视口上进行绘制的,而不是在窗口部件本身上绘制。像往常一样,我们显式地启用反走样,因为无法假定渲染提示属性(render hint)的默认值是什么。

我们遍历每一项,并获取每一项的模型索引以及该项基于视口坐标系统的矩形区域。如果这个矩形区域是无效的(不应该如此),或者它在视口中是不可见的,这就是说,它的下边框在视口之上,或它的 y 坐标在视口之下都不进行绘制。

对于那些要绘制的项,首先获取由基类提供的 `QStyleOptionViewItem` 选项对象。然后设置该选项对象的矩形区域为项的矩形区域。使用 `QRectF::toRect()` 方法把 `QRectF` 对象转换成了 `QRect` 对象,并且在该项被选中或是当前项的情况下相应地更新选项对象的状态。

最重要的是,我们不自己绘制这些项!而是交由视图的委托——可能是基类的内置 `QStyledItemDelegate` 或由类的使用者设置的一个自定义委托来绘制这些项。这是为了确保该视图支持自定义委托。

这些项是一行一行地进行绘制的,把它们压到一起尽可能地利用可用的空间。但由于每一项的文本可能包含不止一个单词,我们需要帮助用户让他能明显地把项区分开。而这通过在每一项周围绘制一个轮廓线来实现。

```
void TiledListView::paintOutline(QPainter *painter,
                                const QRectF &rectangle)
{
    const QRectF rect = rectangle.adjusted(0, 0, -1, -1);
    painter->save();
    painter->setPen(QPen(palette().dark().color(), 0.5));
    painter->drawRect(rect);
    painter->setPen(QPen(Qt::black, 0.5));
    painter->drawLine(rect.bottomLeft(), rect.bottomRight());
    painter->drawLine(rect.bottomRight(), rect.topRight());
    painter->restore();
}
```

轮廓线是通过绘制一个矩形,然后绘制两条线(一条在矩形之下,另一条在矩形右边),提供一种非常细微的阴影效果。

```
void TiledListView::resizeEvent(QResizeEvent*)
{
    hashIsDirty = true;
    calculateRectsIfNecessary();
    updateGeometries();
}
```

如果视图调整了尺寸大小,那么必须重新计算所有项的矩形区域并更新滚动条。前面已经介绍过 `calculateRectsIfNecessary()` 方法,所以这里只需要介绍 `updateGeometries()` 方法即可。

```
void TiledListView::updateGeometries()
{
    QFontMetrics fm(font());
    const int RowHeight = fm.height() + ExtraHeight;
    horizontalScrollBar()->setSingleStep(fm.width("n"));
    horizontalScrollBar()->setPageStep(viewport()->width());
    horizontalScrollBar()->setRange(0,
        qMax(0, idealWidth - viewport()->width()));
    verticalScrollBar()->setSingleStep(RowHeight);
    verticalScrollBar()->setPageStep(viewport()->height());
    verticalScrollBar()->setRange(0,
        qMax(0, idealHeight - viewport()->height()));
}
```

这是个在 Qt 4.4 中引入的保护槽,用于更新视图的子窗口部件,如滚动条。

窗口部件的理想宽度和高度是在 `calculateRectsIfNecessary()` 中计算出来的。理想高度总是足以显示所有的模型数据,而对于理想宽度也是如此(如果视口的宽度足以显示最宽的项)。如前面提到过的,视图部件的真实尺寸是多少并不重要,因为用户永远只能看到视口里的内容。

这里设置水平滚动条的单步长(`single step size`)(也就是说,当用户点击某一个箭头的时候滚动条会移动多远)为一个字母“n”的宽度,即一个字符。还设置了它的页步长(`page step size`)(也就是说,当用户点击滚动条滑块的左边或右边的时候滚动条会移动多远)为视口的宽度。还设置了水平滚动条的取值范围为从0到窗口部件的理想宽度,而不考虑视口的宽度(因为这个宽度已能看到)。垂直滚动条的设置也是类似的。

```
void TiledListView::mousePressEvent(QMouseEvent *event)
{
    QAbstractItemView::mousePressEvent(event);
    setCurrentIndex(indexAt(event->pos()));
}
```

这是需要实现的最后一个事件处理函数。用它把用户点击和选中的项设置为当前的项。因为我们的视图是一个 `QAbstractItemView` 的派生类, `QAbstractItemView` 本身又是 `QAbstractScrollArea` 的派生类,所以鼠标事件的位置是基于视口的坐标系统的。这不算问题,因为 `indexAt()` 方法所期望传入的 `QPoint` 类型参数正是基于视口坐标系统的。

关于 `TiledListView` 类最后需要注意的一点是它假设用户使用的是从左到右的语言,比如英语。阿拉伯语和希伯来语用户将会发现这个类比较混乱,因为他们使用的是从右至左的语言。我们将使该类在从左到右和从右到左的语言下都能进行工作的修改留给读者,作为一个练习(窗口部件的从左到右或从右到左的状态可通过 `QWidget::layoutDirection()` 来获得;这通常是与 `QApplication::layoutDirection()` 相同的,但最好使用 `QWidget` 的函数以确保严格正确)。

像所有的 Qt 标准视图类那样, `TiledListView` 在数据项和显示项之间也是一一对应的。但在一些情况下我们可能希望把两项或者更多的项以某种方式合并在一起显示出来——但这不被 `QAbstractItemView` API 支持,也不能通过自定义委托来获得。尽管如此,我们依然能产生一个视图完全以我们想要的方式显示数据(如同在下一节中所看到的那样),但在这样做之前先要避开 `QAbstractItemView` API,同时提供我们自己的 API 来替换。

6.2 与模型相关的可视化视图

在这一节中我们将从头做起创建一个视图类作为一个 `QWidget` 的子类,并将提供与 `QAbstractItemView` API 所不同的自己的 API。当然也可以创建一个 `QAbstractItemView` 子类,但因为我们要创建的视图是针对一个特定的模型,并且会把一些项合并起来显示,依照一个不需要的或不相关的 API 来创建似乎没什么意义。

我们将要创建的可视化窗口部件设计为呈现一个人口普查资料表。保存这些数据的是一个表格模型,每一行都包含年度数据、男性人口数据、女性人口数据和总人口数据。图 6.3 展示了 `CensusVisualizer` 应用程序(`censusvisualizer`)的中间区域的界面。截图中有两个表现数据的视图。左边是一个标准 `QTableView`,以传统形式来呈现数据。右边是用一个 `CensusVisualizer` 视图来呈现数据,它是通过按人数比例的、用渐变色填充的彩条来呈现男性人口和女性人口数据的。

我们不能使用 Qt 的 `QHeaderView` 来呈现这个可视化视图的表头,因为已经把两列合并在一起。因为这个原因创建的 `CensusVisualizer` 视图是一个在它内部整合了三个其他窗口部件的 `QWidget`:一个自定义的 `CensusVisualizerHeader` 来提供水平表头,一个自定义的 `CensusVisualizer-`

View 来显示数据,一个 QScrollArea 来容纳 CensusVisualizerView 并提供滚动和调整尺寸大小的功能。这些类之间的关系如图 6.4 所示。

我们首先从应用程序的 main() 函数中这个可视化视图的创建讲起。

```
CensusVisualizer *censusVisualizer = new CensusVisualizer;
censusVisualizer->setModel(model);
```

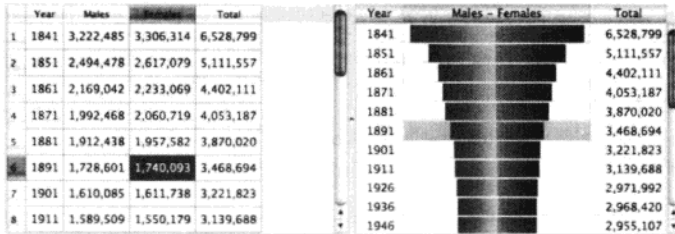


图 6.3 一个 QtTableView 视图和一个 CensusVisualizer 视图

这段代码看上去和工作起来就像我们所希望的那样——先创建了可视化视图,然后调用 CensusVisualizer::setModel() 来提供给视图一个模型。在程序的 main() 函数中的后面部分,创建了 QtTableView 视图,两个视图进行了布局并加了几个信号-槽连接以给程序提供相应的行为表现。我们将略过这些内容,而只专注于这个可视化视图类的设计和编码以及整合进去的表头和视图类。

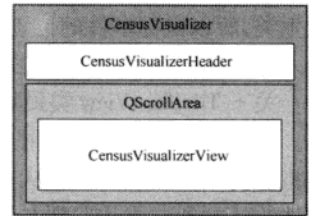


图 6.4 CensusVisualizer 的各个类之间的关系

6.2.1 可视化视图部件

这个可视化视图部件是一个能直接拿来使用的视图,所以先看一下 CensusVisualizer 类,这将给我们一个了解,接下来就要介绍被该视图整合进去的那两个提供外观的自定义类的上下文环境。下面是头文件中 CensusVisualizer 类的定义,但不包括私有数据成员:

```
class CensusVisualizer : public QWidget
{
    Q_OBJECT

public:
    explicit CensusVisualizer(QWidget *parent=0);

    QAbstractItemModel *model() const { return m_model; }
    void setModel(QAbstractItemModel *model);
    QScrollArea *scrollArea() const { return m_scrollArea; }
    int maximumPopulation() const { return m_maximumPopulation; }
    int widthOfYearColumn() const { return m_widthOfYearColumn; }
    int widthOfMaleFemaleColumn() const;
    int widthOfTotalColumn() const { return m_widthOfTotalColumn; }
    int selectedRow() const { return m_selectedRow; }
    void setSelectedRow(int row);
    int selectedColumn() const { return m_selectedColumn; }
    void setSelectedColumn(int column);

    void paintItemBorder(QPainter *painter, const QPalette &palette,
                        const QRect &rect);
    QString maleFemaleHeaderText() const;
    int maleFemaleHeaderTextWidth() const;
    int xOffsetForMiddleOfColumn(int column) const;
    int yOffsetForRow(int row) const;

public slots:
    void setCurrentIndex(const QModelIndex &index);
```



```
signals:
    void clicked(const QModelIndex&);

private:
    ...
};
```

尽管没有显示私有数据成员,但值得一提的是,整合进去的 `CensusVisualizerHeader` 保存为私有成员变量 `header`,另外整合进去的 `CensusVisualizerView` 保存为私有成员 `view`。当然,两个都是指针。该类还有一个指向模型的指针以及一个指向包含了 `CensusVisualizerView` 的 `QScrollArea` 类型的指针。其他的私有成员变量都是整型数据,其中大部分的获取方法(`getter`)都以内联方式实现,在这里列出,其中可写成员的设置方法(`setter`),过一会将进行介绍。

视图用最大人口数来计算男性-女性人口数彩条的最大宽度,以充分利用空间,每当调用 `setModel()` 时就会进行计算。

表头和视图绘制自身时都要用到获取宽度的获取方法(`getter`)。选中的行号和列号要保持记录,并且表头使用它们的值来高亮显示选中的列,视图使用它们的值高亮显示选中的项(或者选中的男性-女性人口数对比项)。

类中还定义了一个信号,这样如果用户通过在视图中点击改变选中项时,就会发射一个 `clicked()` 信号来告知相关的对象。

`CensusVisualizer` 类中的非内联函数包括构造函数以及其他 10 个方法。`paintItemBorder()`、`maleFemaleHeaderText()` 和 `maleFemaleHeaderTextWidth()` 方法供整合进来的表头和视图所使用,所以我们将延后介绍,直到用到它们的时候再进行讲解,但将在这里介绍所有的其他方法。

```
const int Invalid = -1;

CensusVisualizer::CensusVisualizer(QWidget *parent)
    : QWidget(parent), m_model(0), m_selectedRow(Invalid),
      m_selectedColumn(Invalid), m_maximumPopulation(Invalid)
{
    QFontMetrics fm(font());
    m_widthOfYearColumn = fm.width("W9999W");
    m_widthOfTotalColumn = fm.width("W9,999,999W");
    view = new CensusVisualizerView(this);
    header = new CensusVisualizerHeader(this);
    m_scrollArea = new QScrollArea;
    m_scrollArea->setBackgroundRole(QPalette::Light);
    m_scrollArea->setWidget(view);
    m_scrollArea->installEventFilter(view);
    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(header);
    layout->addWidget(m_scrollArea);
    layout->setContentsMargins(0, 0, 0, 0);
    layout->setSpacing(0);
    setLayout(layout);
    connect(view, SIGNAL(clicked(const QModelIndex&)),
           this, SIGNAL(clicked(const QModelIndex&)));
}
```

首先为年度和总人口数列设置固定宽度,该宽度基于预期它能够处理的最大数字并加上一些边距^①。这里设置的总人口数列的宽度只是一个初始默认值;实际的宽度要在 `setModel()` 方法中基于模型中的最大人口数值重新计算。然后创建需要整合的视图和表头部件。尽管将 `this` 作为它们的父对象,因为使用了 `QScrollArea` 来包含视图,所以这个 `QScrollArea` 对象将会被重新设置为视图的父对象。

① 在本书中的惯例是当想要一个水平内距时使用“W”字母,在想要一个字母的宽度时使用“n”字母,例如水平滚动。

QScrollArea 类在 Qt 中与众不同,它没有被设计为可被子类化的。反而它的使用模式是把另一个窗口部件整合到 QScrollArea 对象里面,就像我们这里做的那样。尽管这种方法非常易于使用,如果想使用继承的方法,可以像一些 Qt 内置的类那样从 QAbstractScrollArea 派生一个类。

我们把视图安装为滚动区域的事件过滤器(event filter)——也就是说,每一个到达滚动区域的事件将首先被发送给视图的 eventFilter()方法)。在后面继续介绍 CensusVisualizerView 类的时候就能明白为什么这是必要的了。

除了设置布局内的边距(margin)为 0 以及内部窗口部件的间隔(spacing)为 0 之外,该布局十分传统。这样使得 CensusVisualizer 部件与其他窗口部件看起来比较像,没有多余的边框区域,在 CensusVisualizerHeader 部件和 CensusVisualizerView 部件(被包含在 QScrollArea 部件中)之间没有间隙。

这个信号连接调用稍有些不同,因为它是一个信号-信号连接。这样就建立起了一种关系,当第一个信号发射后其结果是第二个信号也被发射。所以在这个示例中,当用户点击视图(例如选中一项),视图的 clicked()信号转到了 CensusVisualizer 对象中,这将以同样的 QModelIndex 参数紧接着发射一个匹配的 clicked()信号。这就是说 CensusVisualizer 的使用者能连接到 CensusVisualizer 的 clicked()信号而无须关心它的内部实现。这使得 CensusVisualizer 更多的是作为一个封装好的独立组件(self-contained component),但是也有不同情况,例如它把整合进来的那些窗口部件暴露在外面的话就是另一种情形了。

```
enum {Year, Males, Females, Total};

void CensusVisualizer::setModel(QAbstractItemModel *model)
{
    if (model) {
        QLocale locale;
        for (int row = 0; row < model->rowCount(); ++row) {
            int total = locale.toInt(model->data(
                model->index(row, Total)).toString());
            if (total > m_maximumPopulation)
                m_maximumPopulation = total;
        }
        QString population = QString::number(m_maximumPopulation);
        population = QString("%1%2")
            .arg(population.left(1).toInt() + 1)
            .arg(QString(population.length() - 1, QChar('0')));
        m_maximumPopulation = population.toInt();
        QFontMetrics fm(font());
        m_widthOfTotalColumn = fm.width(QString("W%1%2W")
            .arg(population)
            .arg(QString(population.length() / 3, ',')));
    }
    m_model = model;
    header->update();
    view->update();
}
```

当要设置一个新模型时,必须告诉表头和视图来更新它们自己。但首先必须计算出一个合适的人口最大值。我们这样做,首先在数据中找到总人口数的最大值,然后在这个最大值数字的最高位数上加 1,并且把其余位数字都置为 0。例如,如果最大总人口数为 8 392 174,那么最大值就变为 9 000 000。

这里使用的算法非常粗糙,但有效的是:创建一个字符串,第一数字是原数字的第一个数字(最高位)加 1,紧接着是原数字长度减一个 0,并把这个字符串转换为整型。对于 0 我们用一个有两个参数的 QString 构造函数来生成,构造函数接受一个计数(count)和一个字符,返回该字符重复 count 次所组成的字符串。

要注意到无法使用 `model->data(model->index(row, Total).toInt()` 方法获得总人口数, 因为模型碰巧把数据保存为本地化字符串的形式(例如, 在美国和英国是“8,392,174”, 在德国是“8.392.174”)而不是整型。解决方法是使用 `toString()` 方法先从模型中获取数据, 然后再使用 `QLocale::toInt()`——该函数接受本地化的数字数据字符串, 返回整型值。

`QLocale` 类也有相应的 `toFloat()` 和 `toDouble()` 方法, 同样还有其他整数类型的方法(诸如 `toUInt()`), 还有从本地化日期和时间字符串中提取日期和时间的方法。当构造(construct)一个 `QLocale` 对象的时候, 它默认使用应用程序的当前区域设置(locale), 但这可以通过使用带有一个参数的构造函数并且用区域名称(用 ISO 639 标准的语言代码和 ISO 3166 标准的国家代码)作为参数来覆盖, 或者使用两个参数的构造函数, 用 Qt 的语言枚举和国家枚举值做参数。

在构造函数中, 我们初始化了总人口数列的宽度, 但是这里我们能够设置适用于实际数据的宽度。宽度被设置为显示最大数字所需的像素宽度, 再加上两个“W”字母的空隙, 再加上为每三位数字加上个逗号分隔符(或其他分隔符)的空间。

```
const int ExtraWidth = 5;

int CensusVisualizer::widthOfMaleFemaleColumn() const
{
    return width() - (m_widthOfYearColumn +
                     m_widthOfTotalColumn + ExtraWidth +
                     m_scrollArea->verticalScrollBar()->sizeHint().width());
}
```

这个方法为男性-女性人口数列返回一个适合的宽度。它计算出在给定的 `CensusVisualizer` 部件本身宽度、其他两列的宽度、垂直滚动条的宽度以及些许边距后剩下的所允许的最大宽度。这将确保当 `CensusVisualizer` 尺寸发生变化时, 额外的宽度空间总是能给予男性-女性人口数列。

```
void CensusVisualizer::setSelectedRow(int row)
{
    m_selectedRow = row;
    view->update();
}

void CensusVisualizer::setSelectedColumn(int column)
{
    m_selectedColumn = column;
    header->update();
}
```

如果以编程的方式使选中行发生了变化, 视图必须更新自己来正确地进行项的高亮显示。与此相同, 如果选中列发生了变化, 那么表头必须高亮显示选中列的标题。

```
void CensusVisualizer::setCurrentIndex(const QModelIndex &index)
{
    setSelectedRow(index.row());
    setSelectedColumn(index.column());
    int x = xOffsetForMiddleOfColumn(index.column());
    int y = yOffsetForRow(index.row());
    m_scrollArea->ensureVisible(x, y, 10, 20);
}
```

这个槽是作为服务接口提供给使用者的, 这样使用者能够通过信号-槽连接来改变 `CensusVisualizer` 的选中项。

一旦设置了选中行和选中列之后, 就要使它在滚动区域中是可见的。 `QScrollArea::ensureVisible()` 方法接受 x 坐标和 y 坐标以及可选的水平边距和垂直边距(两个默认值都是 50 像素)作为参数。我们减少了边距以在用户点击顶部或底部的可见行时避免不需要的滚动。

这里实际上做了个权衡, 如果垂直边距太大的话, 点击顶部或底部的项将会导致不必要的滚动。如果边距太小, 当用户使用制表键或向下方向键来访问底部的项时, 这个项就不能完全显示出来。

```
int CensusVisualizer::xOffsetForMiddleOfColumn(int column) const
{
    switch (column) {
        case Year: return widthOfYearColumn() / 2;
        case Males: return widthOfYearColumn() +
            (widthOfMaleFemaleColumn() / 4);
        case Females: return widthOfYearColumn() +
            ((widthOfMaleFemaleColumn() * 4) / 3);
        default: return widthOfYearColumn() +
            widthOfMaleFemaleColumn() +
            (widthOfTotalColumn() / 2);
    }
}
```

这个方法用来获取当前列相对应的一个合适的 x 坐标偏移量。这是通过基于列宽计算给定列的水平方向的中点坐标来实现的。

```
const int ExtraHeight = 5;
int CensusVisualizer::yOffsetForRow(int row) const
{
    return static_cast<int>((QFontMetricsF(font()).height()
        + ExtraHeight) * row);
}
```

这个方法用来获取给定行的 y 坐标偏移量,这是通过计算给定行索引乘以行的高度来实现的。

由 `xOffsetForMiddleOfColumn()` 和 `yOffsetForRow()` 方法返回的 x 坐标偏移量和 y 坐标偏移量是假定 `CensusVisualizerView` 正好是能够显示所有数据的尺寸。这种假定是有依据的,因为 `CensusVisualizerView` 强制如此——就如我们将来在 `CensusVisualizerView::eventFilter()` 方法中要看到的那样。这意味着即使视图只有一部分可能显示出来,也不用做任何滚动相关的计算,因为包含了 `CensusVisualizerView` 部件的 `QScrollArea` 已经为我们处理好这个问题了。

现在已经完成了对 `CensusVisualizer` 类的介绍。除了构造函数和 `setModel()` 方法之外,没有多少其他代码。这是因为所有的窗口部件的外观以及大部分行为都被在 `CensusVisualizer` 构造函数中创建并布局的 `CensusVisualizerHeader` 和 `CensusVisualizerView` 的类实例处理掉了。现在就来依次介绍每一个整合进来的类,先从表头开始。

6.2.2 可视化视图的整合表头部件

`CensusVisualizerHeader` 部件为 `CensusVisualizer` 提供了列标题,如图 6.3 所示。因为是亲自绘制的,我们可以借此机会通过使用不同的渐变色填充来赋予它比通常的 `QHeaderView` 看上去更具视觉冲击的三维效果(如果想与 `QHeaderView` 恰好匹配,就应使用 `QStyle` 的方法来绘制)。

头文件中的类定义相当简单;下面是完整的公有 API:

```
class CensusVisualizerHeader : public QWidget
{
    Q_OBJECT
public:
    explicit CensusVisualizerHeader(QWidget *parent)
        : QWidget(parent) {}

    QSize minimumSizeHint() const;
    QSize sizeHint() const { return minimumSizeHint(); }

protected:
    void paintEvent(QPaintEvent *event);
    ...
};
```

构造函数的函数体是空的。唯一实现的方法是 `minimumSizeHint()`、`sizeHint()`、`paintEvent()`，以及两个由 `paintEvent()` 调用的私有辅助方法(稍后将介绍)。

```
QSize CensusVisualizerHeader::minimumSizeHint() const
{
    CensusVisualizer *visualizer = qobject_cast<CensusVisualizer*>(
        parent());
    Q_ASSERT(visualizer);
    return QSize(visualizer->widthOfYearColumn() +
        visualizer->maleFemaleHeaderTextWidth() +
        visualizer->widthOfTotalColumn(),
        QFontMetrics(font()).height() + ExtraHeight);
}
```

列的宽度可以从父对象 `CensusVisualizer` 中获取,所以必须做类型转变(像这里一样,使用 `qobject_cast <>()` 或使用 `dynamic_cast <>()`)来获取父对象的指针,用来访问所需要的数据[如果使用了 `dynamic_cast <>()` 那么编译器必须启用 RTTI(即 Run Time Type Information,运行时类型信息),现今的编译器默认都会打开]。我们需要的最小宽度即所有列的宽度的和,最小高度就是在使用窗口部件字体情形下一个字符的高度再加上一些边距。

`maleFemaleHeaderTextWidth()` 方法以及它所依赖的 `maleFemaleHeaderText()` 方法都是由 `CensusVisualizer` 类提供的,因为整合进来的两个自定义窗口部件都使用了这两个方法。为了完整性把它们显示在这里。

```
int CensusVisualizer::maleFemaleHeaderTextWidth() const
{
    return QFontMetrics(font()).width(maleFemaleHeaderText());
}

QString CensusVisualizer::maleFemaleHeaderText() const
{
    if (!m_model)
        return "- ";
    return QString("%1 - %2")
        .arg(m_model->headerData(Males, Qt::Horizontal).toString())
        .arg(m_model->headerData(Females, Qt::Horizontal)
            .toString());
}
```

`maleFemaleHeaderTextWidth()` 方法返回男性-女性人口数列显示它的标题所需要的宽度,`maleFemaleHeaderText()` 方法则返回标题内容本身。

```
void CensusVisualizerHeader::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHints(QPainter::Antialiasing |
        QPainter::TextAntialiasing);
    paintHeader(&painter, height());
    painter.setPen(QPen(palette().button().color().darker(), 0.5));
    painter.drawRect(0, 0, width(), height());
}
```

`paintEvent()` 方法先进行绘制对象的设置,并把大部分工作交由 `paintHeader()` 方法去做,最后以在整个表头外部绘制一个矩形框结束。

```
void CensusVisualizerHeader::paintHeader(QPainter *painter,
    const int RowHeight)
{
    const int Padding = 2;
    CensusVisualizer *visualizer = qobject_cast<CensusVisualizer*>(
        parent());
    Q_ASSERT(visualizer);
```

```

paintHeaderItem(painter,
    QRect(0, 0, visualizer->widthOfYearColumn() + Padding,
        RowHeight),
    visualizer->model()->headerData(Year, Qt::Horizontal)
        .toString(),
    visualizer->selectedColumn() == Year);

paintHeaderItem(painter,
    QRect(visualizer->widthOfYearColumn() + Padding, 0,
        visualizer->widthOfMaleFemaleColumn(), RowHeight),
    visualizer->maleFemaleHeaderText(),
    visualizer->selectedColumn() == Males ||
    visualizer->selectedColumn() == Females);
...
}

```

这个方法依次绘制每一列的表头。对于每一列它又调用 `paintHeaderItem()` 方法,并把绘制对象、绘制区域、要绘制的文字、该项(也即是该列)的选中状态作为参数传递进去。这里略去了总人口数那一列的绘制,因为它与年度列非常相似。

```

void CensusVisualizerHeader::paintHeaderItem(QPainter *painter,
    const QRect &rect, const QString &text, bool selected)
{
    CensusVisualizer *visualizer = qobject_cast<CensusVisualizer*>(
        parent());

    Q_ASSERT(visualizer);
    int x = rect.center().x();
    QLinearGradient gradient(x, rect.top(), x, rect.bottom());
    QColor color = selected ? palette().highlight().color()
        : palette().button().color();
    gradient.setColorAt(0, color.darker(125));
    gradient.setColorAt(0.5, color.lighter(125));
    gradient.setColorAt(1, color.darker(125));
    painter->fillRect(rect, gradient);
    visualizer->paintItemBorder(painter, palette(), rect);
    painter->setPen(selected ? palette().highlightedText().color()
        : palette().buttonText().color());
    painter->drawText(rect, text, QTextOption(Qt::AlignCenter));
}

```

这个方法是实际上绘制每一个表头项的方法。首先获取到 `CensusVisualizer` 对象的指针,因为要使用到它里面的方法。然后创建了线性渐变(linear gradient),其颜色取决于该项的选中状态。渐变由从中间的浅色开始,逐渐变为顶部和底部的深色,这里使用了比 `QHeaderView` 用的颜色更浅或更深的颜色来产生强对比的三维效果。当渐变对象设置好后,使用它来绘制项的背景。接着围绕该项绘制一个轮廓线——实际上只绘制了两条线,一条在底部,另一条在右边。最后在项中间绘制文本内容。

为完整起见,下面是 `paintItemBorder()` 方法:

```

void CensusVisualizer::paintItemBorder(QPainter *painter,
    const QPalette &palette, const QRect &rect)
{
    painter->setPen(QPen(palette.button().color().darker(), 0.33));
    painter->drawLine(rect.bottomLeft(), rect.bottomRight());
    painter->drawLine(rect.bottomRight(), rect.topRight());
}

```

这里选择使用两条线来绘制轮廓线(outline)因为在本例中它的效果要比绘制一个矩形框更好一些。

关于 `CensusVisualizerHeader` 类现在介绍完毕。这个类出乎意料地简单明了,许多工作仅仅是设置绘制对象和渐变对象并做一些简单的绘制。这与 `CensusVisualizerView` 类形成鲜明对比,就如下一小节将要介绍到的,对于后者我们必须同时实现它的外观和行为。

6.2.3 可视化视图的整合视图部件

自定义 `CensusVisualizerView` 部件被用来显示模型的数据。该窗口部件的尺寸大小是多少无关紧要,因为它被嵌入到了一个 `QScrollArea` 部件中,而 `QScrollArea` 部件能在必要的时候提供滚动条,还能处理一般的滚动相关的事情。这就给了我们自由以便专注于视图部件的外观和行为。下面是头文件中类的定义的公有部分:

```
class CensusVisualizerView : public QWidget
{
    Q_OBJECT
public:
    explicit CensusVisualizerView(QWidget *parent);

    QSize minimumSizeHint() const;
    QSize sizeHint() const;

signals:
    void clicked(const QModelIndex&);

protected:
    bool eventFilter(QObject *target, QEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void keyPressEvent(QKeyEvent *event);
    void paintEvent(QPaintEvent *event);
    ...
};
```

这个类还有几个私有方法,都是用来支持绘制数据的(将会在后面介绍),一个私有数据成员,一个指向 `CensusVisualizer` 父对象的指针。先简要地看一下公有的方法和槽,然后详细过一遍保护的事件处理方法来看一下它们是做什么的以及如何做的,首先来看一下构造函数。

```
CensusVisualizerView::CensusVisualizerView(QWidget *parent)
    : QWidget(parent)
{
    visualizer = qobject_cast<CensusVisualizer*>(parent);
    Q_ASSERT(visualizer);
    setFocusPolicy(Qt::WheelFocus);
    setMinimumSize(minimumSizeHint());
}
```

`CensusVisualizerView` 对象是在 `CensusVisualizer` 类的构造函数中创建的,并把 `CensusVisualizer` 自己作为 `CensusVisualizerView` 对象的父对象传递了进去。尽管如此,还是选择保留一个 `CensusVisualizer` 的成员指针 (`visualizer`) 以访问 `CensusVisualizer` 类,因为在视图 (`CensusVisualizerView`) 构建后,它又随即被传递给了一个 `QScrollArea` 部件——该窗口部件取得了视图的所有权,并成为了视图的父对象(有一种方法可以避免保持成员指针,即通过调用 `qobject_cast<CensusVisualizer*>(parent())->parent()` 来访问可视化部件 `CensusVisualizer`)。

Qt 提供了几种不同的焦点策略: `Qt::NoFocus` (用于标签及其他只读窗口部件)、`Qt::TabFocus` (窗口部件接受制表键切换获得焦点)、`Qt::ClickFocus` (窗口部件在鼠标点击时获得焦点)、`Qt::StrongFocus` (窗口部件接受制表键切换和鼠标点击获得焦点,即 `Qt::ClickFocus` 加上 `Qt::TabFocus`) 和 `Qt::WheelFocus` (即 `Qt::StrongFocus` 再加上接受鼠标滚轮获取焦点)。这里我们使用编辑部件通常使用的是 `Qt::WheelFocus`。

这里省略了 `minimumSizeHint()` 方法的实现,因为它几乎与 `CensusVisualizerHeader::minimumSizeHint()` 完全相同,唯一不同的是在类内部定义了 `visualizer` 数据成员 (`CensusVisualizerHeader` 的父对象是 `CensusVisualizer`,它的父对象并没有重新设置过,所以它不需要一个单独的 `visualizer` 成员变量)。

```

QSize CensusVisualizerView::sizeHint() const
{
    int rows = visualizer->model()
        ? visualizer->model()->rowCount() : 1;
    return QSize(visualizer->widthOfYearColumn() +
        qMax(100, visualizer->maleFemaleHeaderTextWidth()) +
        visualizer->widthOfTotalColumn(),
        visualizer->yOffsetForRow(rows));
}

```

如果已经设置了模型,就要给所有的行留有足够的空间;否则只留一行的空间。`CensusVisualizer::yOffsetForRow()` 返回的 y 坐标是我们所需要的高度,这是因为传入的行号参数等于模型中的行数。对于列则使用 `CensusVisualizer` 类构造时计算出来的固定宽度,再加上计算出来的男性-女性人口数列宽(或 100 个像素,取两者中较大的那个值)。

```

bool CensusVisualizerView::eventFilter(QObject *target, QEvent *event)
{
    if (QScrollArea *scrollArea = visualizer->scrollArea()) {
        if (target == scrollArea && event->type() == QEvent::Resize) {
            if (QResizeEvent *resizeEvent =
                static_cast<QResizeEvent*>(event)) {
                QSize size = resizeEvent->size();
                size.setHeight(sizeHint().height());
                int width = size.width() - (ExtraWidth +
                    scrollArea->verticalScrollBar()->sizeHint()
                    .width());
                size.setWidth(width);
                resize(size);
            }
        }
    }
    return QWidget::eventFilter(target, event);
}

```

`CensusVisualizerView` 被设定为包含它的 `QScrollArea` 对象的事件过滤器。这意味着每一个发送到 `QScrollArea` 对象的事件都要先被送到这个方法中。

我们唯一感兴趣的事件是 `QEvent::Resize`。当这个事件发生时,即当滚动区域的大小发生变化时,也要同步改变 `CensusVisualizerView` 部件的大小。总是设置视图的高度为显示所有数据所需要的高度,设置视图的宽度为当垂直滚动条出现时可用的宽度。也就是说,如果用户滚动了视图并且,点击了某一行,仍可以如同整个窗口部件是可见的一样工作,计算哪一行被点击时不需要考虑滚动。

在一个 `eventFilter()` 实现内部我们可以自由地(至少原则上是)对这个事件做任何处理:可以改变它、替换它、删除它或忽略它。要阻止一个事件进一步走下去(无论我们是否对它进行了处理),或者如果删除掉一个事件,那么必须返回 `true` 来表明这个事件已经处理过了;否则必须返回 `false`。这里我们用到了这个事件,但并不想干涉它的正常行为,所以没有改变它的任何参数,并在最后调用了基类的实现。

```

void CensusVisualizerView::mousePressEvent(QMouseEvent *event)
{
    int row = static_cast<int>(event->y() /
        (QFontMetricsF(font()).height() + ExtraHeight));
    int column;
    if (event->x() < visualizer->widthOfYearColumn())
        column = Year;
    else if (event->x() < (visualizer->widthOfYearColumn() +
        visualizer->widthOfMaleFemaleColumn() / 2))
        column = Males;
}

```

```

else if (event->x() < (visualizer->widthOfYearColumn() +
                      visualizer->widthOfMaleFemaleColumn()))
    column = Females;
else
    column = Total;
visualizer->setSelectedRow(row);
visualizer->setSelectedColumn(column);
emit clicked(visualizer->model()->index(row, column));
}

```

`QMouseEvent::y()` 方法返回鼠标点击位置相对于该窗口部件顶部的 y 偏移量。需要感谢已经把 `CensusVisualizerView` 嵌入到了 `QScrollArea` 对象中,还要感谢它总是有足够的高度显示所有的数据(在 `eventFilter()` 中确保了)。可以直接使用 y 坐标而不管该窗口部件是否已滚动过了。因此这里通过 y 坐标除以行的高度来获得点击所在的行号。

对于列计算,比较 x 坐标偏移量:如果它小于年度列的宽度,那么表明点击了年度列;如果它小于年度列宽加上半个男性-女性人口数列宽,那么表明点击了男性人口数列;依次类推。

一旦知道了行和列,就通知 `CensusVisualizer` 来选中它们,就我们所知,这样做的结果会调用视图和表头的 `update()` 方法,使得正确的行和列能高亮显示。最后发射 `clicked()` 信号,带着一个选中项的模型索引参数(由模型计算出来),这又会相应导致 `CensusVisualizer` 发射它自己的 `cicked()` 信号给与与之相连接的对象,带着同一个模型索引。

```

void CensusVisualizerView::keyPressEvent(QKeyEvent *event)
{
    if (visualizer->model()) {
        int row = Invalid;
        int column = Invalid;
        if (event->key() == Qt::Key_Left) {
            column = visualizer->selectedColumn();
            if (column == Males || column == Total)
                --column;
            else if (column == Females)
                column = Year;
        }
        ...
        else if (event->key() == Qt::Key_Up)
            row = qMax(0, visualizer->selectedRow() - 1);
        else if (event->key() == Qt::Key_Down)
            row = qMin(visualizer->selectedRow() + 1,
                      visualizer->model()->rowCount() - 1);
        row = row == Invalid ? visualizer->selectedRow() : row;
        column = column == Invalid ? visualizer->selectedColumn()
                                   : column;
        if (row != visualizer->selectedRow() ||
            column != visualizer->selectedColumn()) {
            QModelIndex index = visualizer->model()->index(row,
                                                            column);
            visualizer->setCurrentIndex(index);
            emit clicked(index);
            return;
        }
        QWidget::keyPressEvent(event);
    }
}

```

这个事件处理函数用于提供通过键盘的方向键在视图内移动的功能。

在 `CensusVisualizer` 内部我们记录了选中的行和列,但对于男性人口数和女性人口数列,它们在直观上(因此从用户的角度来看)是一列。为了处理这个问题,如果用户点击了向左方向键,并且当前列是男性人口数列或是女性人口数列之一,那么把下一个要移动到的列设置为年度列。如果当前列是年度列,那么什么也不做,如果当前列是总人口数列,那么设置要移动到的

列是女性人口数列。对于向右方向键按键的处理也非常相似(所以略过了这段代码):如果当前列是男性人口数或女性人口数列之一,那么设置要移动到的下一列为总人口数列。如果当前列是年度列,那么设置要移动到的下一列为男性人口数列,如果当前列为总人口数列,那么什么也不做。

如果用户按了向上方向键,那么设置当前行之前的那一行为新的当前行——如果已经是第一行,那么什么也不做。同样相似的是,如果用户按了向下方向键,那么设置当前行之下一行为新的当前行——如果当前行已经是最后一行了,那么什么也不做。

如果新的选中行或选中列或者两者都与当前选中的不同,那么要调用设置选中行和列的方法。这将导致在视图和表头中调用 `update()` 方法,同时还确保了选中项是可见的。我们还发射了一个 `clicked()` 信号,带着选中项的模型索引作为参数。

最后,如果选择了一个新的项,就不能调用基类的实现了,因为我们已经自己处理了这些按键事件而不想它传递到滚动区域中去。这是因为滚动区域会处理方向键,把它解释为对滚动的请求,不是我们所希望或需要的,因为我们自己处理滚动。反之,如果不处理这些按键事件,就要调用基类实现来为我们进行处理。

将这个方法与鼠标事件处理函数比较一下,鼠标事件处理函数在设置选中行和列时不需要确保选中项是可见的——因为用户必须在可见区域才能进行点击。但是这里,用户能够,比如说在最后一个可见行上按下向下方向键,所以必须调用 `QScrollArea::ensureVisible()` (在 `CensusVisualizer::setCurrentIndex()` 中进行的调用)以确保视图滚动到正确的地方。

对于 `Home`、`End`、`PageUp` 和 `PageDown` 按键的支持,它们与用于处理方向键的代码遵循了同样的原则,我们留给读者作为练习(当实现 `PageUp` 和 `PageDown` 的时候,传统习惯是向上或向下滚动可视区域再减一行的高度,以使用户有一行关联内容用来定位)。

前面刚刚介绍的 `eventFilter()`、`mousePressEvent()` 和 `keyPressEvent()` 方法提供了视图的行为。现在来看一下 `paintEvent()` 以及它要用到的几个私有辅助方法,来看看视图的外观到底是怎样渲染出来的。

```
void CensusVisualizerView::paintEvent(QPaintEvent *event)
{
    if (!visualizer->model())
        return;
    QFontMetricsF fm(font());
    const int RowHeight = fm.height() + ExtraHeight;
    const int MinY = qMax(0, event->rect().y() - RowHeight);
    const int MaxY = MinY + event->rect().height() + RowHeight;

    QPainter painter(this);
    painter.setRenderHints(QPainter::Antialiasing|
                           QPainter::TextAntialiasing);

    int row = MinY / RowHeight;
    int y = row * RowHeight;
    for (; row < visualizer->model()->rowCount(); ++row) {
        paintRow(&painter, row, y, RowHeight);
        y += RowHeight;
        if (y > MaxY)
            break;
    }
}
```

这个方法的开始,先计算了几个常量,具体为每行所允许的高度,以及绘制事件中的最小和最大 y 坐标,要减去或添加一行的高度以确保即使只有一部分显示出来的行也会进行绘制。

因为这个窗口部件是在 `QScrollArea` 里的,它的高度总是恰好能显示所有项的,我们不需要计

算偏移量,也不需要自己计算哪些是可见的,哪些是不可见的。然而,为了性能考虑,应该只绘制可见项。

传入的绘制事件参数有一个 `QRect` 类型的属性指定了要重绘的矩形区域。对于小窗口部件来说通常忽略这个区域而是重绘所有内容,但是对于模型-可视化视图部件来说,它可能有大量的数据,为了更有效率,我们只绘制那些需要绘制的内容。所以有了这些常量,我们设置绘制对象,并计算需要绘制的第一行以及该行的 y 坐标[或许把 y 坐标用 $y = \text{MinY}$ 进行初始化比较诱人,但 MinY 并不总是与 $\text{row} * \text{RowHeight}$ 相同,因为在 $\text{MinY}/\text{RowHeight}$ 表达式中整数(如我们需要的)被截断(truncation)了]。

一切就绪,我们遍历模型的那些行,从第一个可见行开始,并且绘制每一行,直到 y 坐标超过了需要重绘的区域,在这时结束绘制。这将确保获得以及绘制至多可见的行数加上两个额外的行,如果模型有成千上万或更多的数据,这将能节省相当大的开销。

```
void CensusVisualizerView::paintRow(QPainter *painter, int row,
                                     int y, const int RowHeight)
{
    paintYear(painter, row,
              QRect(0, y, visualizer->widthOfYearColumn(), RowHeight));
    paintMaleFemale(painter, row,
                   QRect(visualizer->widthOfYearColumn(), y,
                         visualizer->widthOfMaleFemaleColumn(), RowHeight));
    paintTotal(painter, row,
              QRect(visualizer->widthOfYearColumn() +
                   visualizer->widthOfMaleFemaleColumn(), y,
                   visualizer->widthOfTotalColumn(), RowHeight));
}
```

这个方法仅仅用来创建一个合适的矩形区域并调用每列的绘制方法。

```
void CensusVisualizerView::paintYear(QPainter *painter, int row,
                                       const QRect &rect)
{
    paintItemBackground(painter, rect,
                       row == visualizer->selectedRow() &&
                       visualizer->selectedColumn() == Year);
    painter->drawText(rect,
                    visualizer->model()->data(
                        visualizer->model()->index(row, Year)).toString(),
                    QTextOption(Qt::AlignCenter));
}
```

一旦绘制了背景后,就剩下项的文本内容需要绘制了。文本内容从模型中获取到,并在列中居中绘制。

`CensusVisualizerView::paintTotal()` 方法与这个方法非常相似(所以这里没有介绍),唯一的不同是总人口数列(total)是右对齐的。

```
void CensusVisualizerView::paintItemBackground(QPainter *painter,
                                                const QRect &rect, bool selected)
{
    painter->fillRect(rect, selected ? palette().highlight()
                                     : palette().base());
    visualizer->paintItemBorder(painter, palette(), rect);
    painter->setPen(selected ? palette().highlightedText().color()
                       : palette().windowText().color());
}
```

绘制背景和前景时使用哪种颜色,取决于该项是否是选中的。这个方法绘制了背景以及边框,并设置了画笔的颜色,以供调用者绘制文本内容。

`paintMaleFemale()` 方法有些长,所以我们把它分成三部分进行介绍。

```
void CensusVisualizerView::paintMaleFemale(QPainter *painter,
    int row, const QRect &rect)
{
    QRect rectangle(rect);
    QLocale locale;
    int males = locale.toInt(visualizer->model()->data(
        visualizer->model()->index(row, Males)).toString());
    int females = locale.toInt(visualizer->model()->data(
        visualizer->model()->index(row, Females)).toString());
    qreal total = males + females;
    int offset = qRound(
        ((1 - (total / visualizer->maximumPopulation())) / 2) *
        rectangle.width());
}
```

首先找出男性人口数量、女性人口数量,以及它们的和是多少(在前面介绍过使用 `QLocale` 类来从本地化字符串中获取数字)。然后计算完整的彩条(colored bar)应该占用多大宽度,并用这个宽度计算出彩条与边框的左边和右边应该缩进(indent)多少以使得彩条在可用矩形区域内占据正确的大小。

```
painter->fillRect(rectangle,
    (row == visualizer->selectedRow() &&
    (visualizer->selectedColumn() == Females ||
    visualizer->selectedColumn() == Males))
    ? palette().highlight() : palette().base());
```

先绘制背景,颜色取决于该项(男性人口数列或女性人口数列)是否被选中了。

```
visualizer->paintItemBorder(painter, palette(), rectangle);
rectangle.setLeft(rectangle.left() + offset);
rectangle.setRight(rectangle.right() - offset);
int rectY = rectangle.center().y();
painter->fillRect(rectangle.adjusted(0, 1, 0, -1),
    maleFemaleGradient(rectangle.left(), rectY,
        rectangle.right(), rectY, males / total));
}
```

接近尾声时,绘制了项的边框,并调整可用矩形区域到合适大小(可能比原来小一些),这样它才能提供给绘制彩条需要的正确的尺寸和位置。最后使用渐变色填充并绘制彩条(在高度上有很小的缩减),男性人口数部分使用深绿到浅绿(从左到右)的渐变色填充,女性人口数部分使用浅红到深红(从左到右)的渐变色填充。

```
QLinearGradient CensusVisualizerView::maleFemaleGradient(
    qreal x1, qreal y1, qreal x2, qreal y2, qreal crossOver)
{
    QLinearGradient gradient(x1, y1, x2, y2);
    QColor maleColor = Qt::green;
    QColor femaleColor = Qt::red;
    gradient.setColorAt(0, maleColor.darker());
    gradient.setColorAt(crossOver - 0.001, maleColor.lighter());
    gradient.setColorAt(crossOver + 0.001, femaleColor.lighter());
    gradient.setColorAt(1, femaleColor.darker());
    return gradient;
}
```

上面这个方法介绍完就圆满结束了。它创建了一种颜色从深色到浅色,另一种颜色从浅色到深色,并在特定位置两种颜色交汇的线性渐变对象。交汇点是由调用者依据 `males/total` 计算出来的;这将确保男性人口数部分和女性人口数部分的宽度能正确地反映其人口比例。

Qt 还有 `QConicalGradient` 和 `QRadialGradient` 这两个相似的 API。

现在我们已经介绍完了 `CensusVisualizer` 类和做了许多工作的、整合进来的 `CensusVisualizer-`

Header 和 CensusVisualizerView 类。当我们有一个模型并想以某种独特的方式呈现,并且项是以某种方式合并起来显示的,且使用自定义委托或基于 QAbstractItemView API 的自定义视图无法满足需要之时,像这样创建自定义类是完美的方案。

现在我们已经介绍完了 TiledListView 和 CensusVisualizer 类。TiledListView 要简短得多,因为它不用显示任何列标题,而且它能依赖基类来提供一些功能。如果想以一种独特的方式来呈现模型数据,比如图形化方式,或者想把一些模型的项以组合方式呈现,这样一个自定义委托就满足不了我们的需求了,必须使用自定义视图。如果采用 CensusVisualizer 类使用的方法,我们能获得完全的控制,只须实现那些真正需要的特性。然而,如果选择创建一个 QAbstractItemView 派生类,仍然可以获得完全的控制,还会无偿得到一些功能,以及更多的功能有可能可以复用——但是必须重新实现所有的纯虚方法,通常应使用那些在表 6.1 中列出的方法。

本章是奉献给 Qt 的模型/视图架构的四章内容中的最后一章。一般而言,最容易的方法是从使用 QStandardItemModel 开始,子类化 QStandardItemModel(或 QStandardItem)使数据可序列化并可反序列化。然后,如果有需要,可以使用一个自定义模型来代替。同样地,使用一个 Qt 标准视图也是开始显示模型数据的最佳方法,如果有需要自定义外观或需要对项进行编辑,最好的(也是最容易的)方法是使用自定义委托。然而,如果没有标准视图和自定义委托的组合能够以要求的方式可视化数据,我们就必须以本章介绍的任一种方式来创建自定义视图了。



第7章 用 QtConcurrent 实现线程处理

- 在线程中执行函数
- 线程中的过滤和映射

线程非常流行,有时也很有用。也正是因为它的流行,使得很多程序设计人员在并非必要的时候也会使用它,从而产生了许多不必要的复杂程序(可参阅“关于线程的争论”的阴影部分)。值得注意的是,本章只是有关 Qt 线程的基本知识——仅用于说明如何使用 Qt 的线程支持,而不是一个关于线程自身的教程^①。

在深入研究线程之前,有必要停下来思考一下那些由线程处理所带来的问题。绝大部分人都希望使用线程来改进程序的性能,但要实现这一点,处理问题的某些要求可能会与我们编写单线程序时所使用的办法截然不同。

实际上,我们不能完全确定使用多线程就一定能够真正改善程序的性能。例如,如果增加使用线程的数量,使它与系统可用内核的数量成正比,这样做或许还会降低程序的性能,因为所获得的收益会因线程竞争的剧增而消失殆尽。有时候,单线程中最有效的算法在多线程中却不一定有效。因此,如果真的是想改进程序的性能,理想的做法是,使用不同的实现方法,并与它们的性能进行比较后加以分类——使用与用户完全相同的硬件和软件配置。

暂且不管这些,假定线程就是正确的解决方案,Qt 对它提供了丰富的支持。特别是 Qt 4.4 引入了 QRunnable 类和 QtConcurrent 命名空间,两者都是设计用做线程高级 API 的支持,这样程序设计人员就不需要再使用那些由 QThread 提供的低级 API 和相关类了。这些高级 API 大大减少了我们与线程相关的许多责任(尽管有些东西还是需要注意的)。

关于线程的争论

得益于 Java 对线程的内置支持和越来越普及的多核处理器,编写具有线程处理功能的程序在近年来得到了飞速发展。

尽管非常流行,但关于线程的争论则一直存在。它增加了程序的复杂性,与单线程程序相比,它让程序变得更难调试、更难维护。在使用线程期间,处理过程通常也难以分开。而且,由于线程的自身存在开销,或者仅仅是因为在线程编程中更容易出错,使得我们也并不一定可以得到性能的改良。

Sun 的首席程序员 Tim Bray 说到:“现在,最好、最优秀的程序员已经在 Java 和 .NET 中为建立和调试线程处理框架花费了十年时间,越来越多的人开始认识到,喜欢线程并不是件好事情;别去碰它了”(摘自 Tim Bray 博客中题为“Processors”的段落,参见 www.tbray.org/ongoing/When/200x/2008/04/24/Inflection)。并非只有他一个人这么认为。计算机科学的创始人之一 Donald Knuth 也说到:“如果认为整个线程的思想就是个失败,我也不会有丝毫的诧异”(摘自对 Donald Knuth 的采访,参见 www.informit.com/articles/article.aspx?p=1193856)。

^① 对于那些大致熟悉线程而并不熟悉 Qt 线程的读者来说,或许可以先从阅读 Qt 的线程文档 qt.nokia.com/doc/threads.html 或者是阅读《C++ GUI Qt 4 编程》(第2版)一书中的“多线程”一章而有所收获。

这里似乎有两个主要问题。第一个问题是,要使用线程,通常就需要程序员增加大量的代码来支持线程功能,这似乎偏离了所要解决问题的初衷。更为糟糕的是,这些代码错综复杂,很难保证其正确性,并且也很难调试。第二个问题是,在硬件层面上,对于并行操作的问题仍然存在很多不同的解决办法,但这些办法都需要编译器的作者们用到不同的技术,而随着时代的进步,这些技术有可能被新的硬件解决办法所代替。

这些也并不都是坏消息。有一种技术或许可以通过高级方式来使用线程,而无需增加程序员的负担,也不需要大量的低级簿记功能(bookkeeping)(即锁定和解锁),这种技术的名称就是软件事务内存(software transactional memory, STM)。可用于 C++ 技术的库正在开发中。Qt 本身提供了 QtConcurrent 函数(会在本章中介绍到)设计用于线程的高级访问功能,它还照顾到了所有的底层细节。其他可行的解决办法还包括 Erlang 和 Go 编程语言,还有苹果公司的 Grand Central Dispatch^①。

当然,还有另外一种方法,可在避免线程大多数缺点的情况下利用多核的优势(但也会丧失线程的部分优点)就是多进程(multi-processing),它有自己的负载集。这需要涉及对单进程的任务分解,例如,会使用到 Qt 的 QProcess 类。然而,这一方法可以降低我们在并发支持方面所冒的危险并减少额外代码,而只需负责处理进程间的全部通信就可以了。

当我们想在一个或多个辅助线程(secondary thread)执行的同时做一些后台处理且无须使用 QThread 所提供的全部功能和灵活性时,就可以使用 QRunnable 类和 QtConcurrent::run()方法,它们很适用于这些情况。这些内容会在 7.1 节中谈到。

QtConcurrent 命名空间还为过滤、映射并简化提供了一些方法——在 7.2 节中谈到这些方法时将会介绍其概念。这些方法特别适合用做有许多项需要处理的情况。遗憾的是,我们不能用这些方法处理 QAbstractItemModel 或 QGraphicsScene 中的项,因为 Qt 不支持模型、场景或它们所含项的锁定;然而,在付出一些内存和处理开销代价的情况下,我们将会看到如何解决这个问题。

有时,使用低级 API 会是不错的方法。下一章将会介绍 QThread 的用法。使用 QThread 可能是 Qt 中使用多线程时最具挑战性的工作之一,但我们将因此获得良好的控制而受益。

在本章的这两节和下一章中,我们会尽可能多地降低线程操作方面的风险和复杂性。要做到这一点,首先就是要完全避免锁的使用,例如,让每个线程都有自己的单独进程。在需要锁定的地方,会尽量减小锁的规模,或把它变成透明,例如,通过创建一些能够处理自身锁的类,以便让客户端不必再对它们自己做任何锁定操作。

Qt 甚至还提供了一些低级类,如 QAtomicInt 和 QAtomicPointer。这些类是创建线程安全数据结构和其他低级线程组件的理想工具,但这些都超出了本书的范围(有关使用它们的思想,可以参阅精选书目中的“*The Art of Multiprocessor Programming*”一书)。

Qt 的线程 API 还包括 QSemaphore、QThreadStorage 和 QWaitCondition。这些类都经常与 QThread 的子类一起联合使用,尽管本章以及下一章都没有用到它们,而是用到了其他的一些类和技术,例如,使用了 volatile bool(参见精选书目中的“*C++ GUI Programming with Qt 4*”一书,其中举例说明了 QSemaphore、QThreadStorage 和 QWaitCondition 类的用法)。

volatile 关键字让变量能够在程序后台得到改变。这就意味着,它将无法被编译器优化(或者

^① Grand Central Dispatch 是由苹果公司在 Snow Leopard 系统上扩展出来的一种多核并行运行机制,目的是帮助开发人员更容易地利用多核处理器的并行处理功能——译者注。

被缓冲!)。当存在于内存地址中的变量由外部程序改变时,例如,在硬件端口中进行改变时——它就非常有用了。但除此之外,volatile bool 在线程化的程序中也非常有用,在这种程序中,变量的值可能会由一个线程改变,再由另一个线程读取。需要注意的是,虽然在线程化的程序中 volatile 不适用于其他数据类型(甚至没有 int),因为可能会存在不同线程同时对同一数据类型进行更新成两种数据类型的情况,但对 bool 则是安全的^①。除 volatile bool 之外,本书中有关线程的例子还用到了 QMutex、QMutexLocker、QReadWriteLock、QReadLocker 和 QWriteLocker,对于 QtConcurrent 命名空间中的函数,则用到了 QFuture 和 QFutureWatcher。

一般来说,当建立和运行单独线程的付出比把任务分配给多核或多处理器所得到的好处多时,线程化就是比较划算的解决方案。因此,除了之前在实现并行算法时所见到的那些明显效果之外,应用线程还是非常有效的,至少在一个地方(也许还有多个地方)可以全部或者绝大部分独立地完成那些相对复杂的位运算。

在 GUI 环境中使用线程是它的另外一个重要用途。如果有需要执行的复杂运算并希望避免用户界面的冻结,或许会通过创建一个处理这种运算的辅助线程来解决。对于网络,则并不需要做,因为 Qt 已经可以不同步地处理网络访问,但对于我们自己的进程,某些时候使用一个或者更多的辅助线程还是非常有效的。在某些情况下,一种可替换的轻便选择是使用本地事件循环,就像在第 2 章中看到的那样。

7.1 在线程中执行函数

当所要处理的项的数量非常少,但每一项的处理过程却很麻烦时,可让一个单独执行的线程只处理一项,这种做法通常会比较方便,也可以保证用户界面的响应。同样,如果有许多项需要处理,则可以先把它们分组(或放到一个工作序列中),然后再把处理过程分解给一些独立的线程,这样或许会更奏效些。

把需要处理的数据分配给一些线程或进程来分别执行的方法主要有四种:可以利用 QProcess 类执行独立的进程(例如,运行多个自包含“工作者”程序的副本);可以利用 QtConcurrent::run() 在 Qt 全局线程池(thread pool)的辅助线程中运行函数或方法;可以创建一些 QRunnable 对象并在 Qt 全局线程池辅助线程中运行它们;或者可以创建一些 QThread 对象并把它作为辅助线程加以运行。本节中,我们将看到如何使用 QtConcurrent::run() 和 QRunnable,在下一章将使用 QThread。

QtConcurrent::run() 函数比较简单:创建一个用来处理数据的函数或方法,并把它传递给 QtConcurrent::run() 函数进行执行。如果想要使用多个辅助线程,可以多次传递同样的函数(通常带有不同的参数)。QRunnable 的用法与此相似。创建一个 QRunnable 子类,把全部要处理的数据放进重新实现的纯虚拟函数 run() 中,把尽可能多的辅助线程当做要传递给 QThreadPool::start() 方法的实例即可。

与 QThread 的用法相比,在辅助线程中使用 QtConcurrent::run() 或者 QRunnable 处理数据有两个潜在的缺点。第一,没有对信号和槽的支持,因此也就没有内置的通信条件(例如,用来表明当前进度)。第二,当处理结束时,没有通知提示,因此,如果想要知道处理进度是否结束,我们必须自己查找。克服这两个缺点的方法都很简单——就是提供对停止运行的支持,在本节的后面会谈到这些。

^① 例如,请参阅 Andrei Alexandrescu 的“volatile——Multithreaded Programmer's Best Friend”一文, www.ddj.com/cpp/184403766。

在这一节,我们将创建一个 Image2Image 应用程序(image2image),如图 7.1 所示^①,它会从特定的目录中查找图像文件,并使用特定的文件格式(如. bmp、.tiff等)为每个图像文件创建一个备份。

一个图像的转换过程涉及到了把它以原始格式读取到内存,然后再把它以新格式保存出来,这是一个混合了磁盘输入/输出(I/O)和数据处理的过程。在这个程序中,我们选择在一个或多个辅助线程中完成这个任务,每个线程都有需要用来转换的唯一文件列表。

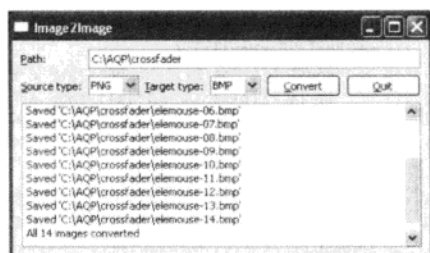


图 7.1 Image2Image 应用程序

既然每个线程都有自己的任务要完成,那么辅助线程之间就没有通信的必要,也就不需要锁定了。当然,还是希望能够把进度通知给主线程(GUI),但我们将通过一些技术来做到这一点,这样做会让 Qt 仅关注于那些与锁相关的必要事情上。

我们已经写出应用程序,以便让它可以使用 QtConcurrent::run() 或是 QRunnable,这取决于 #define。在下一小节中,将介绍如何使用 QtConcurrent::run(),然后再介绍如何使用 QRunnable。在这两个小节中,都将尽可能少地包含那些与用户界面相关的代码,而仅将重点放在线程方面。

尽管它们两者都使用了相同的主窗口框架和取消控制方法,但在与进度进行通信时却对它们使用了不同的方法。对于 QtConcurrent::run(),通过使用一个自定义事件来进行通信。对于 QRunnable,通过启动一个槽来进行通信,并且因为 QRunnable 不是 QObject 子类,在不使用 emit 的情况下,该槽必须要能够得到调用(也就是说,并不是以常见的 Qt 方式来发射信号)。这两种方法的用法纯粹是为了说明两种技术的不同之处;本来可以只需简单地仅使用自定义事件或者仅使用激活槽的方式来完成上述功能。

除了它的方法和窗口部件,主窗口类还有三个私有数据类型的项:

```
int total;
int done;
volatile bool stopped;
```

total 变量存储要处理图像的总数目,done 变量是已经成功转换图片的数目,stopped 布尔值用来通知辅助线程用户是否已经取消操作。值得注意的是,这里对“开始转换”和“取消转换”只使用了一个按钮(即 convertOrCancelButton 按钮)。

一旦选定目录、图片源和目标格式,用户就可以按下 Convert 按钮开始转换工作了。按钮一旦被按下,该按钮就会变成 Cancel 按钮(只改变它的文本),所以,用户可以在任何时候停止转换工作。这个按钮与 convertOrCancel()槽相连。

```
void MainWindow::convertOrCancel()
{
    stopped = true;
    if (QThreadPool::globalInstance()->activeThreadCount())
        QThreadPool::globalInstance()->waitForDone();
    if (convertOrCancelButton->text() == tr("&Cancel")) {
        updateUi();
        return;
    }
    QString sourceType = sourceTypeComboBox->currentText();
```

^① 顺便提一下,用来输入路径的行编辑器使用 QCompleter 来缩小用户必须键入的有效路径的范围——在第9章会涉及到这部分内容。

```

QStringList sourceFiles;
QDirIterator i(directoryEdit->text(), QDir::Files|QDir::Readable);
while (i.hasNext()) {
    const QString &filenameAndPath = i.next();
    if (i.fileInfo().suffix().toUpper() == sourceType)
        sourceFiles << filenameAndPath;
}
if (sourceFiles.isEmpty())
    AQP::warning(this, tr("No Images Error"),
                 tr("No matching files found"));
else {
    logEdit->clear();
    convertFiles(sourceFiles);
}
}

```

这个槽从设定 `stopped` 变量值为 `true` 开始,通知那些运行中的任意辅助线程都必须停止。然后,检查是否还有辅助线程在 Qt 的全局线程序列中运行,如果有的话,那么就一直等待(阻塞),直到所有全局线程运行完成。

`QThreadPool::globalInstance()` 方法会向 Qt 的全局对象 `QThreadPool` 返回一个指针,而 `QThreadPool::activeThreadCount()` 则会返回线程池中正处在激活工作状态的线程的数目。当然,这个值有可能会是 0。`QThreadPool::waitForDone()` 方法会等待线程池中的所有线程都完成任务,这样它就可能很长时间地阻塞用户界面。为了避免这个问题的出现,必须确保所有的线程都在等待发生之前就完成任务,在本应用程序中,通过把 `stopped` 变量值设置为 `true` 就可以解决这个问题。

如果用户已经取消,只需调用 `updateUi()` 方法(在此未做介绍)把 Cancel 按钮的文本改成 Convert 并返回,因为我们已经停止了所有的辅助线程。

如果用户点击了 Convert,就会创建一个所选目录中文件的列表,这些文件的后缀名会和所选的源类型相匹配。如果列表为空,通知用户,然后返回。如果列表不为空,清除日志(一个只读的 `QPlainTextEdit` 对象),并以文件列表为参数,调用 `convertFiles()` 实施转换操作。

`convertFiles()` 有两种实现方法,一种是使用 `QtConcurrent::run()`,将会在 7.1.1 节中介绍,另一种是使用 `QRunnable`,将会在 7.1.2 节中介绍。在这两种情况下,我们都将介绍 `convertFiles()` 方法和支持框架方法。

7.1.1 QtConcurrent::run() 的使用

`QtConcurrent::run()` 函数的参数包含一个函数、一个或多个传递给函数的可选参数,它会在 Qt 全局线程池中的一个辅助线程中执行该函数。其形式为:

```
QFuture<T> run(Function, ...)
```

Function 必须是一个指向函数(仿函数 functor)的指针,函数返回 T 型对象。省略号(...)代表一个变量参数列表(也就是说,0 个或者更多的可选参数)。如果给出这些参数,当被 `QtConcurrent` 调用时,会把它们传递给 Function,因此,如果传递了任何参数,这些参数都必须要与 Function 的形式相匹配。

现在回到 `image2image` 应用程序,看一个 `QtConcurrent::run()` 实际调用的例子。

```

void MainWindow::convertFiles(const QStringList &sourceFiles)
{
    stopped = false;
    updateUi();
    total = sourceFiles.count();
}

```

```
done = 0;
const QVector<int> sizes = AQP::chunkSizes(sourceFiles.count(),
    QThread::idealThreadCount());

int offset = 0;
foreach (const int chunkSize, sizes) {
    QtConcurrent::run(convertImages, this, &stopped,
        sourceFiles.mid(offset, chunkSize),
        targetTypeComboBox->currentText());
    offset += chunkSize;
}
checkIfDone();
}
```

从设定 `stopped` 变量为 `false` 开始,然后调用 `updateUi()` 方法(在此未做介绍)通过改变它的文本来把 `Convert` 按钮变成 `Cancel` 按钮。把 `total` 变量的值设置成列表中文件的数目,`done` 变量设置成 0,因为还没有进行任何文件的转换。

应当创建一个转换单个图像文件的函数,每次都用这个函数和需要处理的文件名作为参数调用 `QtConcurrent::run()`,潜在地创建出与列表中文件数量一样多的辅助线程。

本可以创建一个函数来转换一个单独的图像文件,为每个必须要处理的文件通过函数和文件名调用 `QtConcurrent::run()` 一次,这样也就潜在建立和列表中文件数目一样多的辅助线程。对于一些非常大的文件,这样的方法或许奏效,但对于非常多却不知大小的文件来说,建立如此多的线程(特别是在 Windows 平台上)所付出的代价好像会与任务分散给辅助线程处理所能得到的潜在收益不成比例。

值得庆幸的是,`QThread::idealThreadCount()` 方法提供了计算程序运行所在平台上支持的辅助线程的最佳数目——考虑到了操作系统、处理器的数量和机器拥有的处理核的数目。对于只有一个处理器、一个处理核的机器,这个方法的返回值或许为 1,对于拥有多个处理器和处理核的机器,返回值则会相应增大。这个数字不一定会与需要处理的文件的数目完全匹配,因此,需要把任务进行划分,这样的话,每个辅助线程(假设使用的辅助线程数目大于 1)都能得到一个和需要处理的文件数目相等的数值(当然,用文件数目来划分任务或许不是在所有情况下都是最好的方法,例如,在一个数目为 20 的文件列表中,前 10 个文件很大,后 10 个文件很小)。

为了把任务分给最合适数量的辅助线程,我们从调用 `AQP::chunkSizes()` 函数(在此未做介绍,但包含在本书的 `aqp.h` 源代码模块中)开始,该函数会在容器中给定项的数目(此处为文件数量)和期望的文件块数(这里,理想的数目就是辅助线程的数目),会返回一个块大小累加的矢量,而这些块大小的值则很有可能会相等。例如,给定的文件列表的数目为 97 项,块大小为 1(1 个辅助线程),应当得到的矢量为 [97];如果给定的块大小为 2,应当会得到的矢量为 [49, 48];如果给定的块大小为 3,则会得到 [33, 32, 32],以此类推。

只要得到了块大小的矢量,在其上进行迭代(如果理想的线程数为 1,只迭代一次,因为只有一个块大小),并为每个块大小调用 `QtConcurrent::run()`。传递的参数有 5 个:一个是 `convertImages()` 函数(将要运行的函数),其他 4 个参数将在调用 `convertImages()` 的时候传给它。这些剩下的参数是:一个是指向主窗口(`this`)的指针,用来与进程进行通信;一个是指向 `volatile stopped` 布尔值的指针,用来观察是否取消处理进程;一个是从全部任务列表中得到的唯一文件列表;最后一个是目标文件类型的后缀名。对 `QtConcurrent::run()` 的每个调用都是非阻塞型的,一旦对它进行调用,就会在辅助线程中用其所含的参数来调用 `convertImages()` ①。

① 如前所述,在线程间使用 `volatile bool` 是安全的,但该技术对其他数据类型则不起作用。

`QStringList::mid()` 方法(实为 `QList<T>::mid()`)带一个 `offset` 和一个可选的 `count` 参数,返回一个 `offset` 型的 `count` 项子列表——如果 `offset + count` 的和大于项的数目,或者没有给定 `count`,则全部使用 `offset` 的项。

一旦所有的辅助线程得到启动,就会调用 `checkIfDone()` 槽(很快将会介绍到),这个槽会进行轮询以确定数据处理是否完成。如果用 `QFutureWatcher<T>` 检测到辅助线程已经完成, `QtConcurrent::run()` 函数就会返回一个 `QFuture<T>`。在下一节,我们将看到如何处理由 `QtConcurrent` 各函数返回的 `QFuture<T>`,但此处将忽略 `QtConcurrent::run()` 的返回值和轮询状态,而仅用来说明一个使用了轮询的例子。

```
void convertImages(QObject *receiver, volatile bool *stopped,
                  const QStringList &sourceFiles, const QString &targetType)
{
    foreach (const QString &source, sourceFiles) {
        if (*stopped)
            return;
        QImage image(source);
        QString target(source);
        target.chop(QFileInfo(source).suffix().length());
        target += targetType.toLower();
        if (*stopped)
            return;
        bool saved = image.save(target);
        QString message = saved
            ? QObject::tr("Saved '%1'")
              .arg(QDir::toNativeSeparators(target))
            : QObject::tr("Failed to convert '%1'")
              .arg(QDir::toNativeSeparators(source));
        QApplication::postEvent(receiver,
                                new ProgressEvent(saved, message));
    }
}
```

这个函数会在一个或多个辅助线程中得到调用,每次调用都带一个唯一的需要处理文件的列表。在每次重复操作(载入并存储图像文件)之前,它都会检测用户是否已经取消了操作——如果已取消,函数就会返回,它正执行的线程也会转变成非激活态。

数据处理的过程非常简单:对于列表中的每个图像文件,创建一个 `QImage` 对象(构造函数将会读入给定的图像文件),然后为目标图像创建一个合适的名称,再使用对象的名称保存图像。`QImage::save()` 方法返回一个简单的成功或者失败的布尔型标志。

`QDir::toNativeSeparators()` 静态方法的参数是由路径或者是路径和文件名组成的字符串,返回值是由 `QDir::separator()` 分隔后的目录字符串(例如,在 Windows 上分隔符为“\”,而在类 UNIX 系统上,分隔符则为“/”)。在源代码中,使用“/”会更方便一些,因为无论是在何种平台上,Qt 都能理解它,所以在字符串中就不需要对它进行转换。但是,当我们想为用户显示路径时,最好还是根据应用程序运行时所在平台的正确形式来显示它。

我们还想告诉用户正在处理的文件的进度。最简单也是最好的方法就是在调用的窗口部件中启用一个槽,下一小节将会介绍它的内容。但此处使用稍微麻烦一点的办法,传递一个自定义事件,仅用来说明该过程是如何完成的。

从创建一个信息字符串开始,然后在自定义事件中把它与成功标志一起传递给接收对象(本例中为主窗口)。`QApplication::postEvent()` 方法会获取这个事件的所有权,因此,就不用担心删除它了。

实际上,有两种方法可用于事件的发送:`QApplication::sendEvent()` 和 `QApplication::postEvent()`。

sendEvent() 方法会立即发送事件,但如果要发送全部事件,就需要小心使用。例如,在线程化程序中,sendEvent() 会在发送者的线程中对事件进行处理,而不是在接收者的线程里进行事件处理。另外,也不能对事件进行压缩或者重排序。例如,多个着色事件就不能压缩成一个着色事件。同时,sendEvent() 不会删除事件,因此,实际应用中会在栈(stack)上创建 sendEvent() 事件。

postEvent() 方法会向接收者的事件序列中添加事件[该事件应当在堆(heap)上使用 new 来进行创建],以便可以让它作为接收者事件循环处理过程中的一部分而得到处理。这就是应当经常用到的技术,因为它与 Qt 的事件处理配合得非常好。

在任何一种情况下,我们都没有注意一个事实,那就是事件会从一个线程切换到另外一个线程——Qt 为我们完美地处理了这种情况。

```
struct ProgressEvent : public QEvent
{
    enum {EventId = QEvent::User};

    explicit ProgressEvent(bool saved_, const QString &message_)
        : QEvent(static_cast<Type>(EventId)),
          saved(saved_), message(message_) {}

    const bool saved;
    const QString message;
};
```

这里给出的是自定义的 ProgressEvent 的完整定义。给予每一个自定义事件一个唯一的 ID(QEvent::User, QEvent::User + 1 等)是非常重要的, ID 的类型为 QEvent::Type, 这样做是为了避免事件之间的相互混淆。我们把事件定义成 struct, 并将布尔型的 saved 标志和 message 文本设置成可公开访问。

```
bool MainWindow::event(QEvent *event)
{
    if (!stopped && event->type() ==
        static_cast<QEvent::Type>(ProgressEvent::EventId)) {
        ProgressEvent *progressEvent =
            static_cast<ProgressEvent*>(event);
        Q_ASSERT(progressEvent);
        logEdit->appendPlainText(progressEvent->message);
        if (progressEvent->saved)
            ++done;
        return true;
    }
    return QMainWindow::event(event);
}
```

如果要在一个特定窗口部件中能够探测并处理自定义事件,就必须重新实现 QWidget::event()。此处,如果数据处理正在进行(也就是说,还没有取消),获取自定义的 ProgressEvent, 会把事件的信息文本添加到 QPlainTextEdit 日志中,如果存储成功,就增加已完成转换的文件数。程序还会返回一个 true 来表明事件已经得到了处理,以便让 Qt 删除该事件,而不是继续寻找另一个能够处理该事件的处理器。但是,如果处理过程已经停止,对于任何的其他事件,都把任务传递给基类的事件处理器。

```
const int PollTimeout = 100;

void MainWindow::checkIfDone()
{
    if (QThreadPool::globalInstance()->activeThreadCount())
        QTimer::singleShot(PollTimeout, this, SLOT(checkIfDone()));
    else {
        QString message;
        if (done == total)
            message = tr("All %n image(s) converted", "", done);
```

```

else
    message = tr("Converted %n/%l image(s)", "", done)
        .arg(total);
    logEdit->appendPlainText(message);
    stopped = true;
    updateUi();
}
}

```

在 `convertFiles()` 槽的结尾处会调用这个槽来开始轮询 (`QObject::tr()` 的用法将在后面讨论到, 参阅“三参数形式的 `tr()` 的用法”的阴影部分)。无论任何时候, 只要 `convertImages()` 函数结束, 就可以使用一个自定义事件或者是启用一个信号, 但仍必须检查活动的线程数目, 以确定它们是否全部结束, 因此, 轮询基本上没有什么优越性。一种替代的方法就是保留 `QtConcurrent::run()` 调用返回的 `QFuture<T>`, 当每个辅助线程都已经完成时, 就用 `QFutureWatcher<T>` 通知我们 (在 7.2 节中将介绍如何使用这种方法)。

在这里, 我们从检查是否所有的辅助线程都在运行来开始。如果还有运行的辅助线程, 创建一个单触发计时器, 每 100 ms 就调用一次这个槽。否则, 如果所有的线程都已经结束, 那么就向日志中添加一个适当的信息, 重置 `stopped` 变量 (如果是因为用户的取消而使所有线程全部结束, 那么这个变量的值也已经变成 `true`), 并更新用户界面 (也就是将 `Cancel` 按钮的文本变成 `Convert`)。

```

void MainWindow::closeEvent(QCloseEvent *event)
{
    stopped = true;
    if (QThreadPool::globalInstance()->activeThreadCount())
        QThreadPool::globalInstance()->waitForDone();
    event->accept();
}

```

为了获得安全清除, 对于多线程应用程序来说, 最好在终止程序之前停止所有的辅助线程。我们已经通过重新实现程序的 `closeEvent()` 做到了这一点, 可以确保在允许终止动作继续之前让任何活动的线程先结束掉。

7.1.2 QRunnable 的使用

替代 `QtConcurrent::run()` 的用法是创建一个 `QRunnable` 子类, 并在 Qt 全局线程池中的一个线程中执行它。在这个小节中, 我们将介绍上一小节中 `convertFiles()` 方法的另一种替代实现方法, 还有一些必要的支持方法, 并通过介绍来了解它们的实现细节。

```

void MainWindow::convertFiles(const QStringList &sourceFiles)
{
    stopped = false;
    updateUi();
    total = sourceFiles.count();
    done = 0;
    const QVector<int> sizes = AQP::chunkSizes(sourceFiles.count(),
        QThreadPool::idealThreadCount());

    int offset = 0;
    foreach (const int chunkSize, sizes) {
        ConvertImageTask *convertImageTask = new ConvertImageTask(
            this, &stopped, sourceFiles.mid(offset, chunkSize),
            targetTypeComboBox->currentText());
        QThreadPool::globalInstance()->start(convertImageTask);
        offset += chunkSize;
    }
    checkIfDone();
}

```



`convertFiles()` 方法的这个版本与我们看到的前一个版本在结构上是一样的。其关键差异在于,对于要处理的文件的每一个块,都创建了 `ConvertImageTask` 对象(`QRunnable` 的一个子类)来处理数据(赋予此对象的参数与 `QtConcurrent::run()` 中传递给 `convertImages()` 方法的参数相同)。可运行对象一旦创建完毕,就对它调用 `QThreadPool::start()`——这会把可运行对象的拥有权赋给 Qt 的全局线程池,并可以让它开始运行。

当可运行对象结束时,线程池会把它删除,这也正是我们想要的效果。在某些情况下,如果必须由我们自己负责可运行对象的删除时,可以通过调用 `QRunnable::setAutoDelete(false)` 来阻止自动删除的发生。

```
class ConvertImageTask : public QRunnable
{
public:
    explicit ConvertImageTask(QObject *receiver,
                              volatile bool *stopped, const QStringList &sourceFiles,
                              const QString &targetType)
        : m_receiver(receiver), m_stopped(stopped),
          m_sourceFiles(sourceFiles),
          m_targetType(targetType.toLower()) {}

private:
    void run();
    ...
}
```

以下是 `ConvertImageTask` 的定义,不包括我们忽略掉了的私有成员数据。通过把它的 `run()` 方法私有化,可以阻止类的子类化,但也阻止了 `run()` 在实例中的调用(因为 `run()` 本应该只能由 `QThreadPool::start()` 调用)。

```
void ConvertImageTask::run()
{
    foreach (const QString &source, m_sourceFiles) {
        if (*m_stopped)
            return;

        QImage image(source);
        QString target(source);
        target.chop(QFileInfo(source).suffix().length());
        target += m_targetType;
        if (*m_stopped)
            return;
        bool saved = image.save(target);

        QString message = saved
            ? QObject::tr("Saved '%1'")
              .arg(QDir::toNativeSeparators(target))
            : QObject::tr("Failed to convert '%1'")
              .arg(QDir::toNativeSeparators(source));
        QMetaObject::invokeMethod(m_receiver, "announceProgress",
                                   Qt::QueuedConnection, Q_ARG(bool, saved),
                                   Q_ARG(QString, message));
    }
}
```

这个方法在结构上与前面看到的 `convertImages()` 函数一致。唯一的不同在于不是通过调用自定义事件来与进度进行通信,而是通过启动主窗口中的一个槽(`MainWindow::announceProgress()`)来实现这一功能的。

既然 `QRunnable` 不是 `QObject` 的子类,它就不会内置支持信号和槽。一种解决办法是对 `QObject` 和 `QRunnable` 进行多重继承,但如果真要实现对信号和槽的支持,一个不错的办法就是创建一个 `QThread` 子类,因为它是一个 `QObject` 子类,拥有数个有用的标准内置信号和槽。另外一种解决方法就是像我们在前一小节中那样使用自定义事件。

我们在这里的选择就是用 `QMetaObject::invokeMethod()` 方法启动一个槽。这个方法带的参数包括：一个接收者、一个要调用的槽的名字、连接的类型（`Qt::QueuedConnection` 对于辅助线程来说是最好的，因为它可以像 `QApplication::postEvent()` 一样使用事件序列）和一些要发送的参数。每一个参数都必须用 `Qt` 的 `Q_ARG` 宏进行定义，该宏带一个类型和一个值。`QMetaObject::invokeMethod()` 方法可以传递的参数可多达 9 个，也可以定义一个返回值，尽管只有在使用 `Qt::DirectConnection` 时返回值才有意义。

由于槽的启动放在了主事件（GUI）的序列上，槽的执行就需要在 GUI 线程中，而不是在 `QMetaObject::invokeMethod()` 调用的辅助线程中。在辅助线程中发射信号也是真的，因为从底层来看，`Qt` 会把信号从辅助线程变成事件。

一些程序员认为使用 `QMetaObject::invokeMethod()` 要比发送自定义事件好一些，因为它可以与 `Qt` 的信号和槽机制无缝对接，并且不需要创建自定义的 `QEvent` 子类或者是重新实现事件要发送到的窗口部件中的 `QWidget::event()`。实际上，线程之间的信号和槽是通过 `Qt` 的事件机制来实现的，但我们不必知道或者是关心 `QMetaObject::invokeMethod()` 的有关用法，而只需享受调用方法所带来的便捷性，而不是去创建自定义事件。

```
void MainWindow::announceProgress(bool saved, const QString &message)
{
    if (stopped)
        return;
    logEdit->appendPlainText(message);
    if (saved)
        ++done;
}
```

这个方法会把给定的信息添加到日志中，如果保存完成，还同时会更新已完成文件的数量——如果进程停止，那么什么也不做。

需要用来支持 `ConvertImageTask` `QRunnable` 剩余部分的结构与使用 `checkIfDone()` 槽来查看所有进程是否结束的 `QtConcurrent::run()` 轮询的结构是一样的，并且当用户使用 `closeEvent()` 终止程序时，也可用来确保所有的辅助线程都已经全部结束。

尽管在之前实现进度通报的 `QtConcurrent::run()` 版本中使用过自定义事件，而在 `QRunnable` 版本中使用过槽调用，但实际上，在这两个版本中，也可以都使用自定义事件，或是都使用槽调用。一般来说，最好使用槽调用，因为它更方便，所需的代码也更少。对于 `QRunnable`，可以像我们在这里一样利用轮询来监测进度，但对于 `QtConcurrent::run()`，就可以使用轮询或者 `QFutureWatcher<T>`（下一节将会介绍 `QFuture<T>` 和 `QFutureWatcher<T>`）。

`QtConcurrent::run()` 和 `QRunnable` 的主要区别在于：`QtConcurrent::run()` 会返回一个 `QFuture<T>`——它提供了一种跟踪（并控制）处理进度的方法，在下一节介绍返回 `QFuture<T>` 的其他 `QtConcurrent` 函数时，还会介绍这些内容。把这些与 `QRunnable` 进行对比，在 `QRunnable` 中，必须自己动手构建监测和控制进度的功能。

当需要用到许多辅助进程时，使用 `QtConcurrent::run()` 或 `QRunnable` 都是非常有用的，诸如处理一些较大的项时，或者像在这里和文件块中一样，都需要处理很多的项。有些时候，尽管有许多项要处理，但在块中处理起来并不方便。为此，`Qt` 提供了其他一些 `QtConcurrent` 函数，下一节将对它们进行介绍。

7.2 线程中的过滤和映射

对于有很多数据项需要以同样的方式进行处理的情况，`QtConcurrent` 命名空间中的函数会是比较理想的选择。对于较少数量的项（或者说，每处理少于 5 项），可以为 `QtConcurrent::run()`、

QRunnable 或者 QThread 创建一个函数来分别处理每一个项。但当有很多项需要处理时(或许有成千上万的项或者更多),那么为每个项都创建一个线程将可能导致大量的开销,这样来依次处理数据或许能更快些。正如在前一节中所看到的那样,一种解决办法就是创建少量的辅助线程,并让每个线程只处理一组项。但在某些情况下,我们确实想单独地处理每一个项,那么 QtConcurrent 就可以对此提供支持了。

QtConcurrent 模块提供的函数有 4 种:过滤器、映射器、简化器(本节将介绍这 3 种)和前一节中介绍过的一个运行函数。过滤器、映射器和简化器的应用情况是给它们一批要处理的数据。把分配的任务留给 Qt,然后在 Qt 的全局线程池的辅助线程处理数据。

过滤和映射的概念来自函数式编程(functional programming)。在那种环境中,一个过滤器就是一个给定了序列的高阶函数(higher-order function,也就是说,把函数作为自己参数之一的函数),过滤器函数会返回一个这些项的新序列,这时过滤器函数会返回 true。一个映射器就是一个带有序列和映射函数的函数,会返回一个新序列,其中的新项通过把映射函数应用到输入序列的对应项后产生。

QtConcurrent 的过滤器和映射器严格遵守函数式编程方法。因此,在 QtConcurrent 中,过滤器会带一个项的序列,过滤器函数会产生一个新的序列,其中仅包括来自原序列的那些返回 true 的项。这意味着,结果序列或许会不包含任何项,或许会包括一些原始项,或许会包括所有的原始项。从概念层面上来说,一个过滤器具有如下类似的工作方式:

```
QList<Item> filter(QList<Item> items, FilterFunction isOK)
{
    QList<Item> results;
    foreach (Item item, items)
        if (isOK(item))
            results << item;
    return results;
}
```

虽然已经使用了 QList,但任意的连续容器,或者是使用开始或结束迭代器定义的容器的一部分,都可以用做要过滤的项目,并且任何的连续容器都可以用做结果。

映射器(不要与 QMap 的容器类相混淆!)带一个项序列和一个映射函数,会产生一个新序列,新序列中的项数会与原始序列中那些应用过映射函数的项(或许与原始项相比会有所不同)的个数完全相同,这些应用过映射函数的项会为结果序列产生出新的项。从概念层面上来说,一个映射器具有如下类似的工作方式:

```
QList<Type2> mapper(QList<Type1> items, MapFunction map)
{
    QList<Type2> results;
    foreach (Type1 item, items)
        results << map(item);
    return results;
}
```

这里的 Type1 和 Type2 可以是相同类型,也可以是不同类型——其中的重点是,映射函数会接收一个 Type1 类型的项,并返回一个 Type2 类型的项。

简化器带一个项序列,并且会把它们简化成单一的项。例如,我们或许有一个数字序列并且想计算出该序列的和或者平均值(average)。QtConcurrent 命名空间中有把过滤与简化相结合、映射与简化相结合的函数。从概念层面上来说,它们有如下类似的工作方式:

<pre>// Filter-Reduce ResultType result; foreach (Item item, items) if (isOK(item)) result.merge(item);</pre>	<pre>// Map-Reduce ResultType result; foreach (Item item, items) result.merge(map(item));</pre>
---	---

在这里把 `isOk()` 作为过滤器函数,把 `map()` 作为映射器函数。结果对象的 `merge()` 方法带有每个项(在映射 - 简化的情况下,或者是处理过的项),并以某种方式进行自我合并。例如:

```
struct ResultType
{
    ResultType() : sum(0) {}
    void merge(int item) { sum += item; }
    int sum;
};
```

如果 `ResultType` 用于数字序列,在映射 - 简化(假定处理函数是取整函数, `int identity(int x) { return x; }`)之后,结果会以 `result.sum` 的形式出现。

当然,让 `QtConcurrent` 函数比循环简单得多的原因在于,不像这里给出的那样,对每一项进行按序处理,而是在一个或多个辅助线程中进行数据处理,这样,就可以并行地处理多个项。还需要明确的是,对轻量级的处理过程付出开销或许并不值得,但对于重量级处理过程, `QtConcurrent` 函数可以有效提高数据的吞吐量。

`QtConcurrent` 命名空间提供了两个版本的过滤、映射和简化函数,即阻塞型和非阻塞型(多线程)。阻塞型版本适用于在辅助线程内使用(也就是说,在 `QRunnable` 或 `QThread` 的子类中),或者是适用于只关心函数行为而不关心阻塞状况的情形。而本章要考虑的则正是无阻塞型版本。

在撰写这些内容时,Qt 的文档表明,当使用 `QtConcurrent` 的时候不需要锁定。处理不影响其他任何东西的独立数据项时,确实是这样的。但在处理模型或是图形场景中的项时,并不能这样做,因为 Qt 没有提供对模型、场景或是它们所包含项的锁定方法。可以通过逐次创建我们自己的替代项来解决这个问题,实现对数据的并行处理,并逐次地更新模型或场景——当然,提供这些开销要比并行处理这些项所带来的优势更重要。

在这一节中,我们将介绍 `Number Grid` 示例(`numbergrid`)。这个程序会显示一个数字网格,如图 7.2 所示。我们会用预先给定或者随机产生的数字生成一组初始数据,然后通过创建一段 JavaScript 脚本并执行它来动态地填充或是改变网格中的内容(创建的脚本会被每个单元格调用一次,并为每个单元格的值、行数和列数预设变量)。

我们决定在辅助线程中对所有数字网格中的数据进行处理。为了做到这一点,无论什么时候,只要想处理项,就必须逐次地为我们想要处理的模型项创建替代项,然后向 `QtConcurrent` 函数给定这些替代项,从而并行地处理这些数据,最后,逐次迭代结果中的替代项来更新模型项。通过使用独立的替代项,避免了模型、窗口或是模型项的任何锁定需求(无论如何都不能把其中的一个进行锁定)。

当需要把一个用户定义的 JavaScript 脚本应用于项时,虽然与刚才所描述的步骤相同,但我们仍然想要追踪错误发生的数量,并追踪(唯一的)错误信息。这需要所有辅助线程都能更新错误总数和信息,因此,我们就需要用到锁定。正如即将看到,我们将在一个小型类中抽象出所有锁定方面的细节,以便让客户端代码可以读取总数和错误信息,而无须自己再做任何明确的锁定声明。

`Number Grid` 使用了 `QtConcurrent` 命名空间函数中的过滤、过滤 - 简化和过滤 - 映射。特别是,它使用过滤来选择网格中所有符合某种标准的单元格;使用过滤 - 简化来计算符合某种标准的单元格的数量;并对网格中的每一个单元格(或每一个选中的单元格)都使用了 JavaScript 脚本映射。

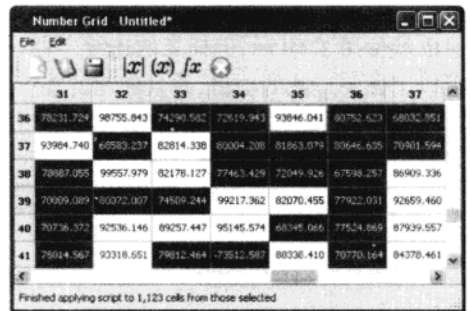


图 7.2 `Number Grid` 应用程序

在 Number Grid 例子中,使用 QtConcurrent 函数来计数并选择单元格或许比直接计数和选择单元格还要慢,因为这些操作都比较简单,因此这些特殊用法仅在描述如何使用 QtConcurrent 过滤和过滤-简化函数时才会用到。然而,在把一个脚本应用到所有或是那些选中的单元格的情况下,使用 QtConcurrent 的映射用法或许要比我们依次对每个相关单元格应用脚本的做法要快得多。如果每个单元格的处理任务都很复杂并且有很多需要处理的单元格,那么潜在的速度提升能力就会更明显了。这是因为,建立和运行线程的开销或许要比并发处理两个或者多个单元格的能力小得多。

Number Grid 就是一个普通的主窗口风格的 C++/Qt 应用程序。它提供了传统的 File 菜单,其中含有 New, Open..., Save, Save As... 和 Quit 选项。我们不会再多介绍这些东西,因为它们都是标准的。

Edit 菜单中的选项为 Count..., Select..., Apply Script 和 Stop,所有这些都用到了 QtConcurrent 的函数。如果用户选择计数(count)或选择(select)功能,应用程序会弹出一个设定应用标准的对话框。对于计数功能,无论是对那些拥有标准值为真(true)的单元格进行计数,还是仅对那些已选中的单元格进行计数,用户都需要定义一个数值关系运算符("<","<=","=",">",">"和"~=",即约等于)并给定一个数字量。对于选择功能,用户可以定义一个数值化关系运算符和一个数字量,这些标准就会应用到网格的所有单元格上。对于应用脚本的功能,会向用户提供一个可以输入任意 JavaScript 脚本的对话框,脚本中会预定义全局变量 cellValue, cellRow 和 cellColumn。然后,用户就可以选择是把脚本应用到网格的所有单元格上,还是仅应用到选中的单元格上。

执行编辑动作所需调用的槽有 editCount(), editSelect() 和 editApplyScript()。我们将依次介绍每一个槽,然后是这些类、方法或是槽所使用的函数,以便可以对如何使用 QtConcurrent 函数有一个全面的了解。虽然 QtConcurrent 函数处理的数据理应都是独立的,但实际上,在一种情况下,使用 QMutex 的确是允许我们读取一些共享数据的。在了解这些槽本身代码之前,先看一下这里给出的从主窗口头文件的定义中取出的代码片段,给出了一些自己的私有数据:

```
QStandardItemModel *model;  
QFutureWatcher<SurrogateItem> selectWatcher;  
QFutureWatcher<Results> countWatcher;  
QFutureWatcher<SurrogateItem> applyScriptWatcher;  
MatchCriteria countCriteria;  
bool applyToAll;  
mutable bool cacheIsDirty;  
QString script;  
ThreadSafeErrorInfo errorInfo;
```

我们认为, QStandardItemModel 比较常用——它被用做将所有网格的数字保存在一个 QStandardItem 子类中的表格模型,这个子类以 double 格式将数字存储在 Qt::EditRole 中(在第3章介绍过 Qt 的表格模型)。

我们将继续看到,当调用一个无阻塞(线程化的) QtConcurrent 函数时,它立即返回一个 QFuture<T>, QFuture<T>代表了计算的结果。然而,返回的 QFuture<T>从没有被正常使用过(至少不是直接使用到)因为计算才刚刚开始。实际上,如果我们试图在计算结束之前访问 QFuture<T>,访问将被阻塞,直到计算完成为止。一旦计算结束, QFuture<T>将保存预期的一个(在简化的情况下)或多个(在过滤和映射的情况下)结果。

无阻塞使用 QtConcurrent 函数的常规模式是把返回的 QFuture<T>传递给 QFutureWatcher<T>。未来观察器追踪 QtConcurrent 函数的进度并在计算结束(或被暂停、恢复或取消)时发射信号来说明到底是哪一个或是哪几个结果准备就绪了(在过滤和映射的情况下会产生一个结果的序列)。未来观察器还提供了一个 API,通过它可以控制 QtConcurrent 函数的行为。例如,对于 pause(), resume() 和 cancel() 的使用。

值得注意的是,虽然过滤和映射产生了结果序列, `QFuture < T >` 使用的类型 `T` 却总是一个单独结果项的类型,在 `QFutureWatcher < T >` 使用的类型 `T` 必须总与未来被观察到的那个结果所使用的类型一致。

在这里,每个程序支持的 `QtConcurrent` 操作都有一个未来观察器。对于选择单元格和对网格中的单元格应用脚本, `QFutureWatcher < SurrogateItem >` 会被重新调用。我们不能在模型中直接并行地处理项,因为模型或是它的项不能被锁定,所以,必须使用替代方法来代替它,后面将看到此情况。对于计数功能,我们有 `QFutureWatcher < Results >`; `Results` 是一个简单的 `struct`,用于保存计数值和总量值,后面将看到它。

对于计数和选择,让用户设定标准来定义哪些单元格应该被计数,哪些该被选中。以下是 `MatchCriteria` 结构体的定义和它所使用的枚举值:

```
enum ComparisonType {LessThan, LessThanOrEqual, GreaterThan,
                    GreaterThanOrEqual, ApproximatelyEqual};

struct MatchCriteria
{
    ComparisonType comparisonType;
    double value;
    bool applyToAll;
};
```

私有数据成员 `countCriteria` 声明的类型为 `MatchCriteria`,用来保存计数的标准。用来做选择和应用的脚本条件保存在 `SurrogateItem` 中,这一点会在后面看到。在进行选择时,会忽略 `applyToAll` 的布尔值。

还有一个布尔成员变量 `MainWindow::applyToAll`;如果该变量的值为 `true`,那么用来进行计数或者应用的脚本将会应用到网格的全部单元格上,否则,仅应用到那些选中的单元格上。

`cacheIsDirty` 布尔值用来跟踪网格数据是否改变过。这是一个可变量,因此可以用于 `const` 方法中。我们将在后面看到它的应用情况。

如果用户选择了 `Apply Script...` 选项, `script` 字符串中就会保存要在每个单元格(或每一个选中的单元格)上执行的 `JavaScript` 脚本。在执行脚本时,有可能会发生错误。我们不想为每个单独的错误都弹出一个警告信息框,因为可能在处理成千上万的单元格时,每个单元格都会发生一个错误。所以,会仅保存一个已发生错误数的计数和一个唯一的错误信息列表。因为任何 `QtConcurrent` 使用的线程都会更新错误数和错误信息,所以就提供一个锁定机制,以便可以实现对它们的依次访问,从而确保在任何一个时间段内,仅允许一个线程更新错误数和错误信息。

要做到逐次访问错误信息,方法之一是创建三个变量,即 `int errorCount`、`QStringList errorMessages` 和 `QMutex errorMutex`。那么,在需要访问错误信息的时候,就可以锁定互斥量(`mutex`)。这种方法就意味着我们必须自己完成所有的簿记功能(`bookkeeping`)。我们选择的是另外一种方法,创建了 `ThreadSafeErrorInfo` 类。这个类为读取和更新错误信息提供了一些方法,并在其内部拥有自己的互斥量,可以对自身进行锁定。这就是说,使用该类的实例的用户不必再去担心锁定的问题,因为它是可以自动得到处理的。以下是从 `ThreadSafeErrorInfo` 的定义中抽取的一段代码:

```
class ThreadSafeErrorInfo
{
public:
    explicit ThreadSafeErrorInfo() : m_count(0) {}
    ...
private:
    mutable QMutex mutex;
    int m_count;
    QSet<QString> m_errors;
};
```

表面上看,这个类并不比我们感兴趣的拥有两个数据项的结构体外加一个互斥量多多少。下面是它的一些方法,用以说明如何使用互斥量。

```
QStringList errors() const
{
    QMutexLocker locker(&mutex);
    return QStringList::fromSet(m_errors);
}
```

QMutexLocker 在它锁定互斥量之前,拥有一个指向互斥量和块的指针。当它超出作用域时,会释放锁定。QList<T>::fromSet() 静态方法会根据设定值产生一个列表。

ThreadSafeErrorInfo 类还有一个 count() 方法和一个 isEmpty() 方法。count() 方法会返回已发生(包括重复发生)的错误信息的个数;而如果错误数为 0,isEmpty() 方法就会返回 true;它们两个与 errors() 方法的结构完全一样(所以也就不再给予介绍)。

```
void add(const QString &error)
{
    QMutexLocker locker(&mutex);
    ++m_count;
    m_errors << error;
}
```

这个方法会增加错误的计数值,也可以把新的信息添加到信息集中。由于信息存储在信息集中,如果要尝试增加一条重复的信息,那么就会直接放弃操作而不做任何通知。

一种替代方法是把信息存储在 QStringList 中,而在 errors() 方法中,复制列表,对该副本调用 QStringList::removeDuplicates(),然后再返回该副本。然而,这就意味着我们或许需要把成千上万条重复的错误信息放在内存中,而不仅仅是把 m_errors 集中的那个唯一的错误信息放在内存中。

```
void clear()
{
    QMutexLocker locker(&mutex);
    m_count = 0;
    m_errors.clear();
}
```

这个方法用来清空数据,例如,为新的 QtConcurrent 函数的调用做好准备。

为 ThreadSafeErrorInfo 类使用 QMutex 非常有效——但在这种情况下,它是要使用的最具效率的锁定类吗?当我们锁定一个互斥量时,也就阻止了其他的任何访问,包括只读性质的访问。在 ThreadSafeErrorInfo 类中,有一些仅需要读取访问,还有一些需要写访问,这样,如果没有线程写操作,允许数个线程并发使用多个读取访问的方法也就没有任何问题了。

要区分读取和写访问,我们需要用 QReadWriteLock 代替 QMutex。然后,在读取访问方法(count()、errors() 和 isEmpty())中,使用 QWriteLocker;而在写访问方法(add() 和 clear())中,使用 QWriteLocker。在后面将看到一个使用了 QWriteLocker 的例子。因此,尽管在这个特殊的示例中,如果仅使用一个辅助线程(例如,在一个处理器、一个处理核的机器上),使用 QMutex 而不使用 QReadWriteLock,不会有什么不同,而在一台拥有超过一个线程数的理想机器上,使用 QReadWriteLock 的性能应当会更高效一些,至少,在理论上是这样的。

拥有为自己提供锁定机制的实例的类称为监视器(monitor),如 ThreadSafeErrorInfo 和 ThreadSafeHash(后面将介绍到)。为了避免死锁,代码在某个锁范围内的监视器方法应当不能调用另外一个锁定的监视器方法。当然,监视器方法中应用的所有锁在方法返回之前必须是未被锁定的——使用 Qt 的 QMutexLocker(或者使用 QReadLocker 或 QWriteLocker 也可以)^①可以轻松实现这一目的。

^① 关于创建监视器的更多细节内容,参见《Qt 季刊》(Qt Quarterly)中名为“Monitors and Wait Conditions in Qt”的文章,它由两部分组成,qt.nokia.com/doc/qq/qq21-monitors.html。

目前,我们已经完成了对私有成员数据的简要介绍,后面将把注意力放在 QtConcurrent 方法的一般使用模式上。先从调用一个 QtConcurrent 函数(比如说, `filtered()`、`mapped()`、`filteredReduced()` 或 `mappedReduced()`) 并把返回的 `QFuture<T>` 存储在变量中开始。然后,调用 `QFutureWatcher<T>::setFuture()`, 把 `QFuture<T>` 作为传递给它的参数。可以通过连接到未来观察器的信号来追踪进度,但至少,需要连接到 `finished()` 信号,以便在进度完成时,能够处理结果,然后就可以访问结果,图 7.3 给出了这种模式的使用方式。

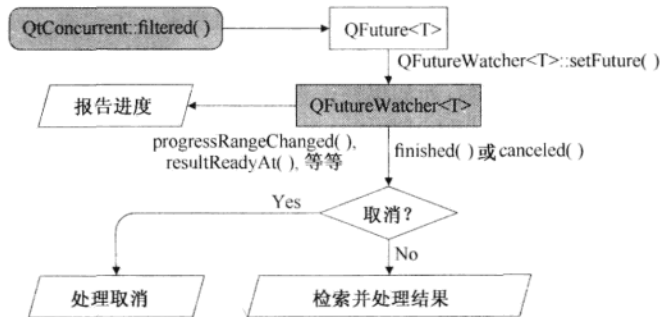


图 7.3 QFutureWatcher 的使用

Number Grid 使用的三个未来观察器都是私有成员变量,起初,我们为它们都创建了两个信号-槽连接。

```

connect(editStopAction, SIGNAL(triggered()),
        &selectWatcher, SLOT(cancel()));
connect(&selectWatcher, SIGNAL(finished()),
        this, SLOT(finishedSelecting()));
  
```

这就是为 `selectWatcher` 最初创建的连接,也为 `countWatcher` 和 `applyScriptWatcher` 创建了同样的连接,不同之处在于它们的 `finished()` 信号是与 `finishedCounting()` 和 `finishedApplyingScript()` 槽相连接的。

用户界面提供了 Stop 菜单项和工具条按钮,因此,如果用户需要,可以随时取消长时间运行的操作。为了满足这种需求,我们把停止动作连接到未来观察器的 `cancel()` 槽。无论一个并行计算是否正常结束还是被迫取消的,总是会发射 `finished()` 信号。

我们很快会看到,无论什么时候,只要未来观察器需要观察“未来情况”,那么就需要创建另外的连接,这样才能利用进度条来为用户显示处理进度,在处理过程中,进度条会叠加在状态栏之上。

在接下来的小节中,我们将介绍如何利用 QtConcurrent 进行过滤、简化和映射,并把重点放在与 QtConcurrent 相关的代码上。

7.2.1 使用 QtConcurrent 进行过滤

有一种方法,可以在一个能够满足由编程或用户给定条件集模型中处理所有这些项,该方法就是使用 QtConcurrent 的过滤函数。QtConcurrent 已提供数个过滤函数,但我们将要用到的是 `QtConcurrent::filtered()`,它具有如下形式:

```

QFuture<T> filtered(Sequence, FilterFunction)
QFuture<T> filtered(ConstIterator, ConstIterator, FilterFunction)
  
```

`Sequence` 是一个诸如 `QList<T>` 或 `QVector<T>` 这样的项的可迭代集合。`ConstIterator` 是开始和结束迭代器,迭代器用来定义 `Sequence` 中的起始点和结束点。`FilterFunction` 用来定义把哪些项放进结果中,把哪些项舍弃掉。它必须具有如下的形式:


```
bool filterFunction(const T&)
```

类型 T 必须与 Sequence 中存储的项的类型相同(也和用于 `QFuture <T>` 的项相同)。这个函数必须为那些放到结果中的项返回 true, 并为那些舍弃掉的项返回 false。

不能直接把 QtConcurrent 函数应用到模型中的项上, 所以, 必须使用前面描述的三个步骤来完成: 第一步, 创建一个替代项的序列, 每一个都与模型中的项相对应; 第二步, 给定 QtConcurrent 过滤函数的条件, 把它应用到替代项的序列上——这样就可以产生一个满足条件的替代项的新序列; 第三步, 在替代项的结果序列上进行迭代, 并相应对每个替代序列中的项更新成等效模型中的项。

在 Number Grid 示例中, 我们想过滤选择所有符合用户条件的模型中的项, 例如, 选择包含值小于 3250 的所有单元格。为了达到这样的效果, 需要用到替代项(surrogate item)——以下是 SurrogateItem 类的完整定义:

```
struct SurrogateItem
{
    explicit SurrogateItem(int row_=0, int column_=0,
                           double value_=0.0)
        : row(row_), column(column_), value(value_) {}

    int row;
    int column;
    double value;
};
```

这个类有些简单, 但却都是精华所在。现在, 我们将会看到如何在现实中实现上面描述的三个步骤, 先从用户选中 Select... 选项启动的 editSelect() 槽开始。

```
void MainWindow::editSelect()
{
    MatchForm matchForm(MatchForm::Select, this);
    if (matchForm.exec()) {
        MatchCriteria matchCriteria = matchForm.result();
        stop();
        view->setEditTriggers(QAbstractItemView::NoEditTriggers);
        QList<SurrogateItem> items = allSurrogateItems();
        QFuture<SurrogateItem> future = QtConcurrent::filtered(items,
            SurrogateItemMatcher(matchCriteria));
        selectWatcher.setFuture(future);
        setUpProgressBar(selectWatcher);
        editStopAction->setEnabled(true);
    }
}
```

首先, 弹出 MatchForm 对话框(在此不做介绍), 获取用户的匹配条件, 也就是用户打算使用的比较运算符和数值。如果用户点击 OK 按钮, 那么就提取匹配标准, 启动选择过程(请参阅之前介绍过的 MatchCriteria 类)。我们从停止所有选择、计数或应用到进度中的脚本开始。然后, 把视图变为只读, 我们不想让用户在处理过程中改变任何值, 因为那样做可能会让计算无效。接下来, 为网格中的每个单元格提取一个替代项列表。

一旦需要处理的替代项准备完毕, 就可以调用 QtConcurrent 函数, 在使用无阻塞型 QtConcurrent::filtered() 函数的情况下, 会向它传递一个要处理的项的序列和一个过滤函数。这个过滤函数实际上就是一个仿函数(functor, 实现 operator()() 的类的实例, 很快将可以看到)。每个项都会调用一次过滤函数(或仿函数的 operator()()), 它会为那些应该进入结果中的项返回 true, 并对那些应该舍弃掉的项返回 false。

QtConcurrent::filtered() 函数会立即返回 QFuture <T>, 并在一个或多个辅助线程中开始进行

数据处理,而只是让该方法得到调用来继续进行数据处理。把返回的 `QFuture < SurrogateItem >` 设定成 `selectWatcher` (类型为 `QFutureWatcher < SurrogateItem >`) 的观察对象;这将会报告进度并提供一种与进程进行交互的方法,例如,暂停、恢复或停止进程。

最后,建立一个进度条来显示数据处理的进度,启用 `Stop` 动作(拥有相应的菜单项和工具条按钮),以便让用户可以在任何时候停止进程。这种情况下,我们并没有提供暂停和恢复功能,但从概念上讲,这种做法与提供了进程停止功能的做法没有区别。

```
void MainWindow::stop()
{
    editStopAction->setEnabled(false);
    if (selectWatcher.isRunning())
        selectWatcher.cancel();
    ...
    if (selectWatcher.isRunning())
        selectWatcher.waitForFinished();
    ...
    editStopAction->setEnabled(false);
}
```

如果用户启动了 `Stop` 动作,就会调用这个方法。我们已经提供了所有情况下都可用的代码(第一行和最后一行),但只有与 `selectWatcher` 有关的部分才会和未来观察器相关,因为这些代码还和 `countWatcher` 和 `applyScriptWatcher` 中的代码相同。我们先从禁用 `Stop` 动作开始,禁用动作会立即向用户提供“程序正在停止”的反馈。停止是一个含有两个步骤的过程:第一步,取消数据处理;第二步,调用 `QFutureWatcher < T >::waitForFinished()` (它用来阻塞进程),用来在我们继续操作之前确保数据处理已真正停止。

这里使用的方法,也就是告诉每一个辅助进程,让它依次停止,然后依次相互等待,在这种情况下非常奏效,因为我们知道,任何时候只能有一个活动的辅助进程(选择、计数或应用脚本)。但在有两个或更多辅助进程执行的情况下,这种方法就意味着需要等待所有停止时间总和那么长的时间(因为我们是线性地依次等待每一个进程的)。随后,将会介绍如何停止多个辅助进程,而让停止时间尽可能地接近停止最慢的线程所用的时间。

```
const QList<SurrogateItem> MainWindow::allSurrogateItems() const
{
    static QList<SurrogateItem> items;
    if (cacheIsDirty) {
        items.clear();
        for (int row = 0; row < model->rowCount(); ++row) {
            for (int column = 0; column < model->columnCount();
                ++column) {
                double value = model->item(row, column)->
                    data(Qt::EditRole).toDouble();
                items << SurrogateItem(row, column, value);
            }
        }
        cacheIsDirty = false;
    }
    return items;
}
```

这种方法用来为网格中每一个单元格创建一个替代项列表。虽然我们返回的列表可能包含有成千上万个项,得益于 Qt 对写时复制(copy-on-write)的广泛应用,实际上,仅会从该方法返回一个左右的有数据价值的指针。

通过对列表的静态化处理(static)来缓冲(cache)调用之间的项,进一步提高本方法的性能(以消耗一些内存为代价),并仅在网格改变时才更新列表(得益于这里没有给出的一个信号-槽连接,无论什么时候,只要模型发射 `dataChanged()` 信号, `cacheIsDirty` 布尔值都会设定为 true)。

```
void MainWindow::setUpProgressBar(QFutureWatcher<T> &futureWatcher)
{
    progressBar->setRange(futureWatcher.progressMinimum(),
                           futureWatcher.progressMaximum());
    connect(&futureWatcher, SIGNAL(progressRangeChanged(int,int)),
            progressBar, SLOT(setRange(int,int)));
    connect(&futureWatcher, SIGNAL(progressValueChanged(int)),
            progressBar, SLOT(setValue(int)));
    progressBar->show();
}
```

当数据处理开始时调用这个函数,并将相关的未来观察器传递给它。在构造函数中创建进度条,把它隐藏并添加到状态栏。这里设定进度条的范围由未来观察器提供,创建两个信号-槽连接,以保证进度条的范围和值都可以处于最新状态。当然,要显示进度,必须要显示窗口部件——这将使它叠加在状态栏上。当介绍 finishedSelecting() 方法时,我们还可以看到进度条会被再次隐藏。

以下是完整的 SurrogateItemMatcher 类:

```
class SurrogateItemMatcher
{
public:
    explicit SurrogateItemMatcher(MatchCriteria matchCriteria_)
        : matchCriteria(matchCriteria_) {}

    typedef bool result_type;

    bool operator()(const SurrogateItem &item)
    {
        switch (matchCriteria.comparisonType) {
            case LessThan:
                return item.value < matchCriteria.value;
            case LessThanOrEqual:
                return item.value <= matchCriteria.value;
            case GreaterThanOrEqual:
                return item.value >= matchCriteria.value;
            case GreaterThan:
                return item.value > matchCriteria.value;
            case ApproximatelyEqual:
                return qFuzzyCompare(item.value, matchCriteria.value);
        }
        Q_ASSERT(false);
        return false;
    }

private:
    MatchCriteria matchCriteria;
};
```

SurrogateItemMatcher 是一个实例为仿函数 (functor) 的类。为得到仿函数实例,类必须要实现 operator()() 方法。对于适合当成“函数”传递给 QtConcurrent 过滤函数的仿函数,必须有公共的 result_type typedef,以用来说明 operator()() 方法的结果的类型。

在创建仿函数实例时,会向构造函数传递要使用的匹配条件(其中包括了比较运算符和数字值)。

无论何时过滤项,QtConcurrent::filtered() 函数都会调用仿函数的 operator()() 方法,并把需要考虑的项传递给它。operator()() 方法会返回一个 bool 值,该值取决于项是否满足了条件。

对于 ApproximatelyEqual (~=) 比较操作,使用 Qt 的全局 qFuzzyCompare() 函数。这个函数可以用来比较两个 float 或者两个 double 值是否大致相等(使用标准的浮点表达式可能会更好些)。

当然,把普通函数当做过滤函数传递是极其有可能的——这样的函数必须能够接受 const T& (也就是一个项)并返回 bool 值。

```

void MainWindow::finishedSelecting()
{
    editStopAction->setEnabled(false);
    progressBar->hide();
    if (!selectWatcher.isCanceled()) {
        view->clearSelection();
        QItemSelectionModel *selectionModel = view->selectionModel();
        const QList<SurrogateItem> items = selectWatcher.future()
                                                .results();

        QListIterator<SurrogateItem> i(items);
        while (i.hasNext()) {
            const SurrogateItem &item = i.next();
            selectionModel->select(
                model->index(item.row, item.column),
                QItemSelectionModel::Select);
        }
        statusBar()->showMessage(
            tr("Selected %Ln cell(s)", "", items.count()),
            StatusTimeout);
    }
    view->setEditTriggers(editTriggers);
}

```

在创建一些未来观察器时,它们每个都有自己的 `finished()` 信号,该信号会连接到相应的槽上,对于 `selectWatcher` 来说, `finished()` 信号就是连接到这个 `finishedSelecting()` 方法上。因此,当过滤过程停止时(无论是因为过滤完成,还是因为用户启用了 `Stop` 动作而被迫停下)都会调用这个槽。

既然进度已经结束,上面的工作就没有什么意义了,先禁用 `Stop` 动作,然后再隐藏进度条。如果处理操作没有取消,那么可以先从清除存在的选择开始。然后,在所有未被过滤掉的替代项(也就是那些满足用户选择条件的所有项)中进行迭代,再对每一个替代项,选择与它相对应的项(替代项的模型索引值可从模型中找到)在视图中,再使用视图的选择模型。

用于状态栏信息上的 `QObject::tr()` 调用有些与众不同。通常情况下,我们只使用单参数的形式。第二个参数是一个用来消除二义性的字符串,这里并不需要。第三个参数是一个数字型计数值,会在“三参数形式的 `tr()` 的用法”的阴影部分加以讨论。

三参数形式的 `tr()` 的用法

`QObject::tr()` 的最常见的使用方法是附带一个参数(即用来转换的文本)或是附带两个参数,这时,第二个参数是一个字符串,用来消除歧义:当需要转换的文本在两个或者更多的地方出现时,就必须根据上下文的不同而进行不同的转换。

三参数形式是在 Qt 4.2 中引入的,用于对数字的处理,其中的第三个参数是一个整型计数值。当使用这种形式的 `tr()` 时,用来转换的文本应当包括的是 `%n`,但它会被计数值所代替(对于本地化的计数值,则为 `%Ln`,例如, U. S. 中的分组逗号)。对于英语来讲,在必须定义为单数或是复数的单词后面加上“(s)”,这是非常有用的。即使不进行转换,文本将会像原来一样显示得很好,例如,“Selected 1 cell(s)”的计数值为 1,虽然这看上去有点业余,但却很容易理解。但如果使用了转换,我们就可以根据是单数还是复数的情况来选择出自己的文本。

Qt 的转换工具非常聪明,它能够为转换人员提供一次机会,让他们有机会为像英语一样的复数化语言(如德语、希腊语、希伯来语和祖鲁语)提供简单的单数和复数形式,还能为那些没有复数化的语言(如阿拉伯语、捷克语、法语、爱尔兰语、毛利语、波兰语和俄语)提供更多的选择^①。

① 有关这方面的全部细节,请参阅《Qt 季刊》(Qt Quarterly)中题为“Plural Form(s) in Translation(s)”的文章, qt.nokia.com/doc/qq/q19-plurals.html。

对于在英语环境下开发的程序,还有一个英语翻译文件,这或许会让人觉得很奇怪,但通常它却非常有用。首先,如果没有提供任何翻译文件,Qt 就会默认成英语文本,这就不需要翻译任何单词,因此,我们只需要翻译三参数的 `QObject::tr()` 调用,就可以保证其他部分不会被翻译。另一方面,部署应用程序后,在用户可见字符处发现打字排版错误或是其他问题,如果错误拥有一个独立的 .qm 文件,那么只需给用户发送一个新的 .qm 文件,就可以轻松改正那些文本,从而去掉受影响的部分。

在 `numbergrid` 应用程序中,用户可见的字符串总数超过了 60 个,我们只需要为其中 4 个提供转换即可。例如,文本 `Selected %Ln cell(s)` 会被转换为 `Selected one cell`(单数)和 `Selected %Ln cells`(复数)。

最后,我们恢复编辑触发器,这样用户就可以再次与网格中的单元格进行交互(编辑触发器的类型为 `QAbstractItemView::EditTriggers`,它们是在创建视图时从中抽取的,保存在一个成员变量中)。

现在,我们已经介绍了如何用带有项序列和过滤函数(在我们的示例中为过滤器仿函数)的 `QtConcurrent::filtered()` 在一个或多个辅助线程中产生一个过滤序列。在接下来的小节中,将介绍如何执行过滤并简化,然后在随后的一小节中,将会介绍如何执行映射。

7.2.2 使用 QtConcurrent 进行过滤并简化

有些 `QtConcurrent` 函数可以用来进行简化操作,包括 `QtConcurrent::mappedReduced()` 和在本小节中将要介绍的 `QtConcurrent::filteredReduced()` 函数,它们具有的形式为:

```
QFuture<T> filteredReduced(Sequence, FilterFunction,
                          ReduceFunction, QtConcurrent::ReduceOptions)
QFuture<T> filteredReduced(ConstIterator, ConstIterator,
                          FilterFunction, ReduceFunction, QtConcurrent::ReduceOptions)
```

与 `QtConcurrent::filtered()` 函数一样,这里的 `Sequence` 是一个可迭代的项集合,如 `QList<T>` 或 `QVector<T>`,而 `ConstIterator` 是开始或结束迭代器,这两个迭代器用来在一个 `Sequence` 中定义一个开始点和结束点。同样, `FilterFunction` 用来定义把哪一项放入到结果中,以及舍弃掉哪一项(前面看到过它的形式)。 `ReduceFunction` 必须具有以下形式:

```
void reduceFunction(R&, const T&)
```

严格来讲,返回类型不一定必须为 `void`,因为它会被忽略掉。非 `const` 型的 `R` 用来存储结果;`T` 的类型与 `Sequence` 和 `QFuture<T>` 中项的类型相同。

`QtConcurrent::ReduceOptions` 是一个枚举类型,在这两种情况下,它的默认值是 `UnorderedReduce|SequentialReduce`,因此,这是一个可选参数。`UnorderedReduce` 部分表明将以 `QtConcurrent` 认为合适的任意顺序来处理项——我们可以强制定义 `Sequence` 的自然顺序,通过使用 `OrderedReduce` 来代替它。`SequentialReduce` 部分表明,在某个时间段将只有一个线程进入 `ReduceFunction`。这就意味着简化函数并不需要是可重入的(reentrant)或是线程安全(thread-safe)的(在撰写这部分内容时,还没有其他可替代的方法,虽然 Qt 的某些未来版本可能会支持并行的简化选项,该选项大概会需要一个可重入的或是线程安全的简化函数)。

过滤并简化与过滤相似,在过滤中,我们必须向 `QtConcurrent` 函数传递一个项序列、一个函数或者是一个决定项是否被计数的仿函数。另外,还必须传递一个可用做结果存储器的函数——这样或许可以把每一个可接受项的某些方面以某种方式进行合并,或是传递一个简单的计数器;此处我们将同时采用这两种形式。

从结构上看,过滤并简化的代码与纯过滤的代码很相似:用一个槽调用进程(此处为 `editCount()`),在进程结束或取消(此处为 `finishedCounting()`)时,调用另一个槽,我们将同时介绍这两个槽,以下是其支持代码:

```
void MainWindow::editCount()
{
    MatchForm matchForm(MatchForm::Count, this);
    if (matchForm.exec()) {
        countCriteria = matchForm.result();
        stop();
        view->setEditTriggers(QAbstractItemView::NoEditTriggers);
        applyToAll = countCriteria.applyToAll;
        QList<SurrogateItem> items = applyToAll ? allSurrogateItems()
                                                : selectedSurrogateItems();
        QFuture<Results> future = QtConcurrent::filteredReduced(
            items, SurrogateItemMatcher(countCriteria),
            itemAccumulator);
        countWatcher.setFuture(future);
        setUpProgressBar(countWatcher);
        editStopAction->setEnabled(true);
    }
}
```

当传递 `MatchForm::Count` 参数时,匹配表允许用户选择一个比较运算符和一个值,还要选择到底是要对所有项进行计数还是仅对选中的项进行计数。如果用户在对话框中点击 OK 按钮,就会提取匹配的条件(`MatchCriteria` 类型)。然后,与前一小节中看到的 `editSelect()` 槽一样,停止所有可能仍处于运行中的进程,把视图设置成只读。

我们注意到,无论进程是应用到所有项,还是只应用到在 `applyToAll` 成员变量中选择的项,都需要知道进程结束时所选的项是哪一个,这样才能显示一个合适的信息。如果用户想要对所有项进行计数,从 `allSurrogateItems()` 方法中抽取替代项序列;否则,可以使用与之非常相似的 `selectedSurrogateItems()` 方法,这会在后面谈到它。替代项与前面用到的替代项相同。

`QtConcurrent::filteredReduced()` 函数会带一个项序列、一个函数或仿函数(此处,我们再次使用 `SurrogateItemMatcher` 仿函数)和一个累加器函数, `itemAccumulator()` (很快就将介绍到)。调用会立即返回一个 `QFuture<Results>`; `Results` 是一个自定义的结构体(struct),在累加器函数中已经创建并使用过它——在介绍累加器时还会介绍它。

最后几行几乎与 `editSelect()` 槽的代码完全相同:向未来观察器传递观察对象,建立进度条,启用 Stop 动作以便让用户可以在想取消的时候取消操作。

```
QList<SurrogateItem> MainWindow::selectedSurrogateItems() const
{
    QList<SurrogateItem> items;
    QItemSelectionModel *selectionModel = view->selectionModel();
    for (int row = 0; row < model->rowCount(); ++row) {
        for (int column = 0; column < model->columnCount();
             ++column) {
            QStandardItem *item = model->item(row, column);
            if (selectionModel->isSelected(item->index())) {
                double value = item->data(Qt::EditRole).toDouble();
                items << SurrogateItem(row, column, value);
            }
        }
    }
    return items;
}
```



这个方法在结构上与 `allSurrogateItems()` 方法相似。唯一的区别在于,它不会为每个项都添加替代项,而是仅为选中的项添加替代项,也就不必缓冲这些项。

在这里,缓冲将消耗一些内存,这在速度上没有什么好处,哪一项进入列表取决于选择模型中的项,这个项会在从一个调用到下一个调用期间很容易发生改变。

```
struct Results
{
    explicit Results() : count(0), sum(0.0) {}

    int count;
    long double sum;
};
```

这个结构体在主窗口的头文件中——它必须在(或包括在)那里,因为我们要在声明未来观察器的 `count` 时用到它: `QFutureWatcher < Results > countWatcher;`。`count` 成员变量用来保存匹配项的个数, `sum` 成员变量用来累加匹配项的总和。

```
void itemAccumulator(Results &results, const SurrogateItem &item)
{
    ++results.count;
    results.sum += item.value;
}
```

这个函数作为倒数第二个参数传递给 `QtConcurrent::filteredReduced()` 函数;我们在之前看到过它的形式。仅对那些已被过滤函数或仿函数接受的项会调用这个函数。在这里,我们简单地增加项的计数值并计算它们的总和。值得注意的是,最初的 `results` 对象(此示例中为 `Results` 类型)通过 Qt 进行创建,因此提供一个可以正确初始化 `struct` 的值的默认构造函数是非常重要的。

对于大量浮点型数字进行累加并计算其总和,这种一次增加 1 的方式就显得有些太过简单和低级了。其中潜藏的问题是,当为两个不同数量级的浮点型数字进行求和时,就会导致精确的丢失;也就是说,如果将一个足够小的数字和一个足够大的数字进行求和,大的数字将不会改变,也就是说,加法“失效”了。即便所有要处理的数字都很小,这也有可能发生(如果数字足够多的话)。最终,累加的总和或许会变得很大,再为其加上很小的数字并不会产生任何变化。有不少方法可以解决这个问题,例如, Kahan 求和算法(参见 en.wikipedia.org/wiki/Kahan_summation_algorithm)。

```
void MainWindow::finishedCounting()
{
    editStopAction->setEnabled(false);
    progressBar->hide();
    if (!countWatcher.isCanceled()) {
        Results results = countWatcher.result();
        QString selected(applyToAll ? QString()
                                     : tr(" from those selected"));
        AQP::information(this, tr("Count"),
                        tr("A total of %Ln cell(s)%2 are %3 %4.\n"
                           "Their total value is %L5.", "", results.count)
                        .arg(selected)
                        .arg(comparisonName(countCriteria.comparisonType))
                        .arg(countCriteria.value)
                        .arg(stringForLongDouble(results.sum)));
    }
    view->setEditTriggers(editTriggers);
}
```

这个函数在结构上与前面看到的 `finishedSelecting()` 方法是相似的。主要的区别在于,它不对结果抽取序列,而是只抽取一个简单的结果对象(这是由 Qt 为我们创建的,并通过累加器函数的调用来对其进行更新)对过滤器所接受的每个项。

`comparisonName()` 函数(在此不做介绍)只是简单地给定的比较类型返回一个 `QString`。例如,给定 `LessThan`,它就返回“<”。

遗憾的是,在撰写这部分内容的时候,还没有一个 `QString::arg()` 方法可以接受 `long double`,因此,我们必须创建自己的函数来对给定的 `long double` 产生一个 `QString`。

```
QString stringForLongDouble(const long double &x)
{
    const int BUFFER_SIZE = 20;
    char longDouble[BUFFER_SIZE + 1];
    int i = snprintf(longDouble, BUFFER_SIZE, "%.3Lf", x);
    if (i < 0 || i >= BUFFER_SIZE) // Error or truncation
        return QString("####");
    return QString(longDouble);
}
```

这个函数使用 `<cstdio>` 模块中的 `snprintf()` 函数。万一出现转换错误或发生截断(truncation),我们选择返回一个电子数据表式(spreadsheet-style)的错误字符串,因而不会引发异常或返回错误的代码。

至此,我们已经完成了对如何使用过滤以及如何使用过滤并简化的介绍。在下一小节中,将会介绍映射。我们不会再单独介绍映射并简化,因为它的工作原理与过滤并简化的工作原理相似,唯一的区别在于,前者会向累加器传送所有(处理过的)项,而不是仅传递那些被过滤函数所接受的项。

7.2.3 用 QtConcurrent 进行映射

映射(mapping)就是把项序列中的每一个项都传送给映射函数的过程,映射函数则依次为接收到的每个项都返回另一个项(或许会是不同的类型)。一共有多个 `QtConcurrent` 映射函数;在这里要使用到的映射函数具有如下形式:

```
QFuture<T> mapped(Sequence, MapFunction)
QFuture<T> mapped(ConstIterator, ConstIterator, MapFunction)
```

与刚才介绍过的过滤并简化函数一样, `Sequence` 是一个可迭代的项集合,如 `QList<T>` 或 `QVector<T>`, `ConstIterator` 是开始或结束迭代器,这两个迭代器用来在 `Sequence` 中定义开始点和结束点。 `MapFunction` 必须具有如下形式:

```
U mapFunction(const T&)
```

`T` 的类型与 `Sequence` 和 `QFuture<T>` 中的项的类型相同, `U` 的类型是处理每一个 `T` 类型项的过程中所得结果的类型——如果想产生一个改进型 `T` 序列,它的类型就会是 `T`。

在这一小节中,我们将会看到如何处理表模型中的所有项。由于过滤的存在,在模型中可以直接对项进行操作,因为并没有锁定它们的办法。因此,不是去创建一个替代项的序列,而是向映射函数传递每个替代项,然后在所有的结果序列上进行迭代,并不断更新模型。

在这个特例中,我们将会把一段由用户创建的 JavaScript 脚本应用到网格中的每一个值上。

```
void MainWindow::editApplyScript()
{
    ScriptForm scriptForm(script, this);
    if (scriptForm.exec()) {
        script = scriptForm.script();
        stop();
        view->setEditTriggers(QAbstractItemView::NoEditTriggers);
        errorInfo.clear();
        applyToAll = scriptForm.applyToAll();
        QList<SurrogateItem> items = applyToAll ? allSurrogateItems()
                                              : selectedSurrogateItems();
        QFuture<SurrogateItem> future = QtConcurrent::mapped(items,
```



```

        SurrogateItemApplier(script, &errorInfo));
    applyScriptWatcher.setFuture(future);
    setUpProgressBar(applyScriptWatcher);
    editStopAction->setEnabled(true);
}
}

```

脚本的来源是一个简单的对话框(在此不做介绍),这个对话框允许用户进入到一些 JavaScript 代码中去。另外,对于标准的 JavaScript 函数和变量,我们将提供三个全局变量:cellValue、cellRow 和 cellColumn,用户可以在他们自己的脚本中使用。我们没有提供网格中任意单元格的值的路径——这样做会增加一定的复杂性,并会让我们偏离本章线程处理这个主题(这也就是为什么程序仅限于一个数字网格而不是一个电子表格的原因)。

如果用户点击 OK 按钮,就可以抽取他们想要应用的脚本。ScriptForm::accept() 方法会使用 Qt 4.5 中引入的 QScriptEngine::checkSyntax() 方法,它可以让用户把有效的脚本仅仅当成脚本而已或者直接点击 Cancel 按钮。

像往常一样,停止仍在继续运行的进程,把视图设为只读。使用之前讨论过的 ThreadSafeErrorInfo 类型的 errorInfo 成员函数变量也可以清除错误的计数值和脚本的错误信息。抽取的是所有项还是选中的项,这取决于用户的请求。QtConcurrent::mapped() 函数带一个项序列、一个映射函数或仿函数。在这种情况下,我们曾用过仿函数,即 SurrogateItemApplier 的一个实例。

映射函数会立即返回,对相应的未来观察器设定它的观察目标。然后更新状态栏,启用 Stop 动作。

这里给出的是 SurrogateItemApplier 类的完整定义:

```

class SurrogateItemApplier
{
public:
    explicit SurrogateItemApplier(const QString &script_,
                                   ThreadSafeErrorInfo *errorInfo_)
        : script(script_), errorInfo(errorInfo_) {}

    typedef SurrogateItem result_type;

    SurrogateItem operator()(const SurrogateItem &item)
    {
        QScriptEngine javaScriptParser;
        javaScriptParser.globalObject().setProperty("cellRow",
                                                    item.row);
        javaScriptParser.globalObject().setProperty("cellColumn",
                                                    item.column);
        javaScriptParser.globalObject().setProperty("cellValue",
                                                    item.value);
        QScriptValue result = javaScriptParser.evaluate(script);
        if (javaScriptParser.hasUncaughtException()) {
            QString error = javaScriptParser.uncaughtException()
                            .toString();
            errorInfo->add(error);
            return item;
        }
        return SurrogateItem(item.row, item.column,
                              result.toNumber());
    }

private:
    QString script;
    ThreadSafeErrorInfo *errorInfo;
};

```

这个类必须提供一个 result_type typedef,这样它的实例才可以被传递给 QtConcurrent 函数。不是

只简单地保存该脚本,而是保存一个指向 ThreadSafeErrorInfo 对象的指针,该对象以前曾被传递给构造函数。

对所有传递给 QtConcurrent::mapped() 函数的项列表中的项都会调用 operator()() 方法。在这种情况下,我们已经选择返回一个相同类型的对象 SurrogateItem,但没有必要这样做——只要使用 result_type typedef 定义了它,就可以返回我们喜欢的任何类型。

先从创建 JavaScript 语法分析器开始,会设置三个全局变量:项的行、列和值。然后,评估 JavaScript 脚本,抽取返回值(脚本中最后表达式的值)。以下给出的是一段简单的 JavaScript 示例:

```
var result = cellValue;
if (cellRow < 10 && cellColumn < 10)
    result *= 2;
result;
```

JavaScript

很明显,这个脚本简单地将最左上角的 100 个单元格的值都乘以 2。

如果脚本有错误,并产生了不可处理的异常,会向 errorInfo 对象增加错误信息。即便可能由多个辅助进程在处理项,我们也不必担心锁定,因为 ThreadSafeErrorInfo 已经替我们完成了这个工作。

最后,我们可以原封不动地返回原始项(如果有未处理的异常),或返回一个新项,它的行和列的值都与原始项相同,把它的数值设定到评估过的返回结果中。

虽然在结构上,finishedApplyingScript() 槽与 finishedSelecting() 和 finishedCounting() 很相似,但我们仍然还会介绍它,以便可以了解脚本错误的处理。

```
void MainWindow::finishedApplyingScript()
{
    editStopAction->setEnabled(false);
    progressBar->hide();
    if (!applyScriptWatcher.isCanceled() &&
        (errorInfo.isEmpty() || applyDespiteErrors())) {
        const QList<SurrogateItem> items = applyScriptWatcher.future()
                                                .results();

        QListIterator<SurrogateItem> i(items);
        while (i.hasNext()) {
            const SurrogateItem &item = i.next();
            model->item(item.row, item.column)->setData(item.value,
                                                         Qt::EditRole);
        }
        QString selected(applyToAll ? QString()
                                     : tr(" from those selected"));
        statusBar()->showMessage(tr("Finished applying script "
                                     "to %Ln cell(s)%1", "", items.count())
                                .arg(selected), StatusTimeout);
    }
    view->setEditTriggers(editTriggers);
}
```

如果进程完成(也就是说,没有取消),并且没有错误,或者用户想要应用脚本而不考虑错误的存在,我们从抽取被处理的项开始(在这个阶段,模型的数据没有改变;所有的结果都在替代项里)。然后在每一个结果项上进行迭代,在模型中,把每一个相应项都设置成新计算过的值。然后输出一个状态信息,告诉用户有多少单元格受到了影响。最后,恢复编辑触发器以便让用户可以再次与网格的值进行交互。

```
bool MainWindow::applyDespiteErrors()
{
    const int MaxErrorStrings = 15;
    QStringList errors = errorInfo.errors();
    if (errors.count() > MaxErrorStrings) {
        errors = errors.mid(0, MaxErrorStrings);
        errors.append(tr("(and %1 others...)")
                     .arg(errorInfo.count() - MaxErrorStrings));
    }
    return AQP::question(this, tr("Apply Script Error"),
                        tr("%1 error(s) occurred:\n%1", "", errorInfo.count())
                          .arg(errors.join("\n")),
                        tr("&Apply Anyway"), tr("&Don't Apply"));
}
```

如果发生了一个或多个错误,这个方法就会得到调用。它弹出一个对话框,可以显示不超过 15 个的错误信息(如果多余 15 个,则提示还剩下多少个)。我们对于将要显示多少错误信息如此小心的原因在于,抽取的网格可能会包含有成千上万个单元格,我们有可能会遇到数千个错误信息,一个对话框肯定远不能显示这么多错误信息。

我们在前面讨论过 `QStringList::mid()` 方法。`AQP::question()` 函数与前面看到的函数类似(例如, `AQP::okToDelete()`)。

现在,我们已经完成了 `QtConcurrent` 函数的介绍。已经介绍了如何过滤、过滤并简化、映射,以及在之前一节中还介绍了运行函数。没有介绍映射并简化,但它的技术与过滤并简化相似,仅是用一个映射函数(或仿函数)代替一个过滤函数(或仿函数)。

使用无阻塞 `QtConcurrent` 函数涉及到一些建立辅助线程的开销,以及在模型或图形场景数据的情况下建立替代项。如果每一项的进程足够复杂,这些开销就可以得到补偿,特别是在有大量的项需要处理的情况下,更是这样。

`QtConcurrent` 是对大量项进行复杂计算的理想选择,有些情况是处理一个或一些拥有复杂处理进程的项,而不牺牲用户界面的反馈性能。一种解决办法是使用 `QtConcurrent::run()` 或 `QRunnable`,我们在前面的部分看到过。但是,如果希望更好地控制 Qt 的信号和槽机制,享受它所带来的便利,使用 `QThread` 或许是更好的选择。

现在,我们已经完成了对 Qt 的高级线程处理类的介绍。它们提供了最简单的通道,让我们在享受线程处理带来的好处的同时将风险最小化。但在某些场合中,我们想要更好地控制程序,并准备好了接受锁定带来的责任,而且要避免死锁。在这些情况下,我们可以使用低级的 `QThread` 类——这正是下一章的主题。



第 8 章 用 QThread 实现线程处理

- 独立项的处理
- 共享项的处理

本章将介绍 QThread 类,该类为线程处理提供了良好的控制,还为 Qt 的信号和槽机制提供了支持。正如前一章所假定的那样,读者应该先了解线程处理的一般知识,并应当了解一些特殊的 Qt 线程处理方面的基础。本章还会假定读者已经阅读了前一章的内容。

如果有少数的项(或是由项构成的少量群组)需要在后台处理,我们希望追踪进度和进度完成情况,通常,最好的办法是创建一个 QThread 子类。Qt 的 QThread 类(还有 QRunnable)是参照 Java 的 Thread 类创建的,因此它们都拥有相同的总体设计思路。例如,都需要用子类来重新实现 run() 方法,都会通过调用 start() 方法来启动线程的执行。

QThread 和 QRunnable 两者之间的关键性差异在于,QThread 是一个 QObject 子类,因此,可以使用单个信号和槽监视进程。实际上,QThread 提供了一些有用的信号和槽,我们可以连接到这些信号和槽。

图 8.1 给出了如何创建和使用多个 QThread 子类实例的示意图。

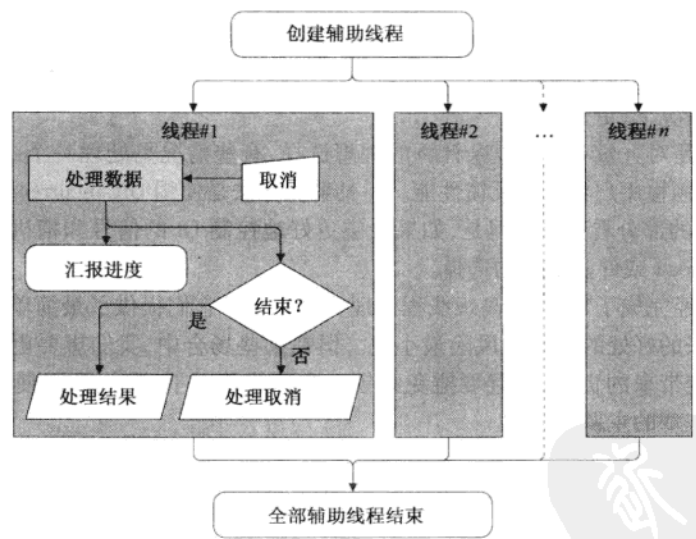


图 8.1 使用 QThread 处理数据

本章将介绍两个不同的应用程序,它们都用到了 QThread,第一个程序用辅助线程来处理独立项,因此它要求没有锁定存在,第二个程序填充了一个共享数据结构,它需要使用锁定以保证访问的安全性。

8.1 独立项的处理

在这一节,我们将介绍 Cross Fader 应用程序(crossfader),如图 8.2 所示。此程序可以让用户选择两幅图像,然后通过建立一个用户定义的中间图像数量来生成一个平滑过渡的图像序列。例

如,用户选择创建三幅平滑过渡的图像,程序最终会生成具有一定比例的5幅图像,其比例为(第一幅图像:第二幅图像):第一幅图像(100:0),平滑过渡图像#1(75:25),平滑过渡图像#2(50:50),平滑过渡图像#3(25:75),第二幅图像(0:100)。

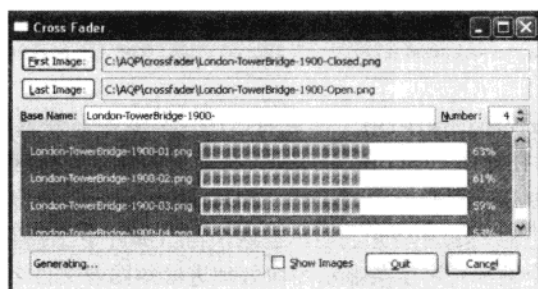


图 8.2 Cross Fader 应用程序

使用了平滑过渡效果的4幅中间图像的结果如图8.2所示,最终结果在图8.3中给出。图8.3显示了第一幅和最后一幅图像,中间四幅为平滑过渡图像,它们所占图像内容比例分别为(100:0)、(80:20)、(60:40)、(40:60)、(20:80)和(0:100)。

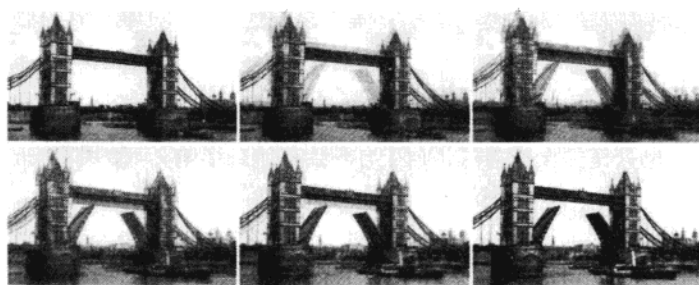


图 8.3 6 幅图像,其中 4 幅为平滑过渡效果

Cross Fader 程序用到了一个单独的 QThread 子类实例,以此来创建每个平滑过渡图像。这是个合理的选择,因为我们限制了可以创建的平滑过渡图像的数量。因此,所有的处理过程都是独立完成的,不需要锁。如果想要并行地创建大量的图像,最好还是使用一些工作线程和一个共享工作序列,将在下一节介绍使用此方法的应用程序。

程序还为每一幅平滑过渡图像创建了一个 QLabel 和一个 QProgressBar,因此用户可以用它们来监视程序的进度。用户可以选择创建任何数量的平滑过渡图像[实际上,我们已经设定微调框(spinbox)所允许的最大值为14],程序会创建用户所选数量的线程、标签和进度条。鉴于标签数量和进度条数量都是变量,我们会在一个 QScrollArea 中将它们显示出来,这样,如果空间不够,则程序会自动提供一个垂直方向的滚动条。

我们已经选择使用一个单独的按钮来实现生成和取消操作,因而它的文本应该是设定成 Generate 还是设定成 Cancel,则需取决于当时所处的情况。另外,如果用户勾选了 Show Images 复选框,就需要打开一个依赖于系统平台的图像浏览器(如果有可用图像浏览器的话),一旦所有图像建立完毕,就可以用该浏览器来显示这些平滑过渡后的图像。

像前一章一样,我们会先介绍那些用来提供上下文环境的基本结构,然后再介绍与线程处理相关的代码,最后介绍用来创建每一幅图像的 CrossFader 的 QThread 子类。

先从枚举变量和程序中 `MainWindow` 类的那些私有成员开始。

```
enum StopState {Stopping, Terminating};

QWidget *progressWidget;

QMap<QString, QPointer<QProgressBar> > progressBarForFilename;
QList<QPointer<QLabel> > progressLabels;
QList<QPointer<CrossFader> > crossFaders;
bool canceled;
```

枚举 `StopState` 用来区分两种停止操作:第一种是因为用户的取消而导致的停止,第二种是因为用户的退出而导致的停止。在介绍 `cleanUp()` 方法时,将谈到为什么要有这两种情况。

`progressWidget` 中放在 `QScrollArea` 内,包含一个 `QGridLayout`。当用户开始生成图像时,程序会创建一系列新的标签和进度条集,并将它们显示在这个窗口部件中。我们把指向进度条的那些指针保存在 `QMap` 中,每一个键都是进度条显示图像处理进度的图像文件名。

此处,我们没有使用普通指针(plain pointer),而是使用 `QPointer` 来保存窗口部件,这些 `QPointer` 是受保护的指针,也就是说,如果指针指向的对象被删除了,这些指针会自动设定为空。就像各种类型的智能指针一样,`QPointer` 要比普通指针复杂得多(它会稍微多用一些内存,或许访问速度也要稍慢一些),但方便之处在于,它可以在使用前检查 `QPointer`,在有可能需要访问已删除对象的情况下,它非常有价值(在前面的“Qt 的智能指针”的阴影部分,我们曾介绍过 Qt 的一些其他智能指针)。

`QWeakPointer` 类是在 Qt 4.6 中引入的,它可以用做普通的弱指针(weak pointer),或相对于 `QObject` 的子类来说,它要比 `QPointer` 更高效一些。然而,`QWeakPointer` 与 `QPointer` 的 API 是不同的——它的便捷性要稍微差一些,但在这里,我们更倾向于使用 `QPointer`,因为与它相关的效率较弱的问题已经被线程的开销完全掩盖掉了(因此不会有什么影响)。使用 `QPointer` 还意味着代码的编译不会因 Qt 的版本是 Qt 4.5 还是 Qt 4.6 而有所改变。

在所有运行的辅助线程中会执行一些动作,`crossfader` 列表可使这件事情变得更为容易,例如,如果用户取消,就可以停止所有动作。在介绍 `cleanUp()` 方法时,我们将介绍为什么要把 `crossfader` 保存在 `QPointer` 中。`Canceled` 布尔值只有在用户界面中会用到,在辅助线程中并不会用到它(因此它不必是 `volatile`)。

只要用户选中两幅图像,Generate 按钮就会变成可用状态(这是因为有一些信号-槽连接并未介绍到)。如果用户点击了这个按钮,它就会调用 `generateOrCancelImages()` 槽,处理过程也就开始了。

```
void MainWindow::generateOrCancelImages()
{
    if (generateOrCancelButton->text() == tr("G&enerate")) {
        generateOrCancelButton->setEnabled(false);
        statusBar->showMessage(tr("Generating..."));
        canceled = false;
        cleanUp();
        QImage firstImage(firstLabel->text());
        QImage lastImage(lastLabel->text());
        for (int i = 0; i < numberSpinBox->value(); ++i)
            createAndRunACrossFader(i, firstImage, lastImage);
        generateOrCancelButton->setText(tr("Canc&el"));
    }
    else {
        canceled = true;
        cleanUp();
        generateOrCancelButton->setText(tr("G&enerate"));
    }
    updateUi();
}
```



在用户点击 `generateOrCancelButton` 后就会调用这个槽,但它的行为取决于按钮是用来启动生成图像,还是取消生成图像。

如果用户已按下 `Generate`,就执行一些与用户接口相关的任务,如状态栏的更新,将 `canceled` 设为 `false` 等。还要调用 `cleanUp()` 以保证没有辅助线程正在运行,删除所有进度窗口部件中显示的标签和进度,并为新的开始做好准备。

然后,为用户想要平滑过渡过的每一幅图像都创建一个 `crossfader`,并将 `generateOrCancelButton` 按钮上的文字变成 `Cancel`。

另一方面,如果用户已经点击了 `Cancel`,可把 `canceled` 设定为 `true`,这样其他主窗口方法就知道生成动作已取消。然后清空,并将按钮重新变成 `Generate` 按钮。

无论发生的是前一种情况还是后一种情况,最后都要调用 `updateUi()` 方法(在此未做介绍);这个方法仅用来让 `Generate` 按钮变得可用或不可用,而 `Generate` 的状态则取决于用户是否已经选中了两个文件名。

像往常一样,键盘加速键是自动设定的,在创建窗口部件后,可以通过在构造函数中调用 `AQP::accelerateWidget(this)` 来实现这一点。但 `generateOrCancelImages()` 方法依赖于对 `Generate/Cancel` 按钮所使用文本情况的了解,同时,我们并不能肯定 `AQP::accelerateWidget()` 会把 `&` 放在何处。解决方法之一是在剔除 `&` 之后再比较按钮上的文本,例如, `generateOrCancelButton->text().replace("&", "") == tr("Generate")`。但我们选择把 `&` 放在这个按钮使用的两个文本中,以保证按钮总是拥有相同的加速键(`Alt + E`),无论它显示的是 `Generate` 还是 `Cancel`。`accelerate *()` 函数会尊重任何手动设置的加速键(有关 `accelerate *()` 函数所提供的 `alt_key`。{hpp,cpp} 模块的更多知识,可参阅“键盘加速键”的阴影部分)。

现在来介绍 `cleanUp()` 方法。在一系列新的图像生成前,或在生成取消时,或在程序终止时,都会调用这个方法。为方便解释,我们分两部分来介绍它。首先,在第一部分,先给出一个不太成熟的实现过程,然后会讨论这一方法所存在的问题,最后再给出一个规避这些问题的实现过程。在完成这些之后,会接着看第二部分,其过程与第一部分的介绍内容一样。

`cleanUp()` 方法的第一部分重点关注的是如何来停止所有正在运行的 `crossfader` 线程。先看一个不太成熟的版本。

```
void MainWindow::cleanUp(StopState stopState)
{
    foreach (CrossFader *crossFader, crossFaders) { // Naive!
        crossFader->stop();
        crossFader->wait();
        crossFader->deleteLater();
    }
    crossFaders.clear();
}
```

这是最为简单的可行办法。遍历所有 `crossfader`,然后对每个 `crossfader` 都调用 `Crossfader::stop()`。这样做几乎不怎么花费时间,因为所有方法都只是简单地把 `bool` 值设置成 `true`。接下来会调用 `QThread::wait()`,因为 `crossfader` 知道它必须结束,以便在一满足 `if (m_stopped)` 声明时它就可以结束了。只要 `QThread::wait()` 的调用有返回值,就可以知道线程已结束,需要让其将自己删除掉。最后,清除 `crossFaders` 列表,因为它现在只包括一些悬摆指针(`dangling pointer`)。

这个方法较为简单,甚至在 `CrossFader` 中都不需要使用 `QPointer`,因为普通指针就已经足够了。但这里就存在一个潜在的重大缺陷:需要依次等待每一个线程的停止,所以,所需等待的时间就是线程停止时间的总和。如果我们准备使用一个稍为复杂的算法,就可以避免这一问题,从而减少停止时间,以使停止的总时间能够接近停止最慢线程的停止时间。

当然,使用这种方法并没有坏处,在实际测试中,复杂方法只有能够产生足以度量的运行优势时才会体现出优势来。即便如此,如果我们仍要使用简单方法,它还是会比使用两个独立的循环要好一些:一个用于停止,一个用于等待和删除。我们会在后面讨论该复杂方法时谈到这些内容。

Cross Fader 程序的源代码包括了这两个方面的执行过程,根据#define 来确定实际执行编译的是哪一个过程。此处,默认使用执行过程中的第一部分:

```
const int StopWait = 100;

void MainWindow::cleanUp(StopState stopState)
{
    foreach (CrossFader *crossFader, crossFaders)
        crossFader->stop();
    while (crossFaders.count()) {
        QMutableListIterator<QPointer<CrossFader> > i(crossFaders);
        while (i.hasNext()) {
            CrossFader *crossFader = i.next();
            if (crossFader) {
                if (crossFader->wait(StopWait)) {
                    delete crossFader;
                    i.remove();
                }
            }
            else
                i.remove();
        }
    }
}
```

此方法所做的第一件事情就是让每个 crossfader 都停止运行。如前所述,对 CrossFader::stop() 的调用是比较快的,它仅仅涉及到把 bool 设定为 true。因此,在循环的最后,每个 crossfader 都知道它一定会停下来。

接下来的一个循环是等待 crossfader 的真正停止,以便可以将其删除掉。为什么要使用两个循环? 因为如果调用 stop() 后再调用 wait() (它会阻塞程序),就会让每个 crossfader 都依次停止,而不是并发停止,这样使得总的停止时间就是所有 crossfader 停止时间的总和。而我们想要达到的效果是,总停止时间接近于最慢停止线程所用的停止时间。鉴于这一原因,我们可以在一个循环中让线程停止,然后在另一个循环中再去等待它们的停止。

我们用于停止线程的算法是,不停地遍历所有的 crossfader,然后依次提取每个 crossfader。由于 QPointer 的存在,如果删除了 crossfader,它就会置空,因此只需简单地从 crossfader 列表中将它移除就可以了(就像在 else 句子中所做的那样)。如果 crossfader 仍然存在,可以对它调用 QThread::wait()。一般情况下,这种方法将“永远”等待下去,但此处只会等待 100 ms。如果线程结束,wait() 将返回 true,此时,就可以删除线程并将它从列表中移除掉。否则,不做任何处理并在下一个循环中继续尝试。

这个方法可有效地给予每个线程 100 ms 的时间片段,因为线程停止就需要这么长的时间。这还意味着,如果一个线程的停止时间较长,不会延长其他线程的等待时间,因为只要超过这个时限,我们就会尝试下一个线程。这就让全部停止时间非常接近于最慢停止线程所用的停止时间,而不是所有线程停止时间的总和。

在 Cross Fader 应用程序中,复杂的算法比简单的算法要稍微好一些。因为 QImage::save() 方法会阻塞程序,并且它也较慢(将磁盘访问和数据处理进行比较),所以全部线程都会把图像保存到磁盘,即便我们已经让它停止了,还是必须等待这些保存的完成。这些时间片段并没有为我们带来所期待的效果。

无论使用哪种算法来停止线程, `cleanUp()` 方法的第二部分都是一样的。现在, 来看一下这一部分的内容。

```
if (stopState == Terminating)
    return;
foreach (QProgressBar *progressBar, progressBarForFilename)
    if (progressBar)
        progressBar->deleteLater();
progressBarForFilename.clear();
foreach (QLabel *progressLabel, progressLabels)
    if (progressLabel)
        progressLabel->deleteLater();
progressLabels.clear();
}
```

一旦所有的线程都已停止, 我们所要做的就是终止程序, 然后返回。但如果准备要清除另一幅图像的生成, 或是清除取消掉的生成操作, 还需要去掉所有用来显示前一个生成进度 (如果有的话) 的标签和进度条。

删除操作非常简单: 遍历所有进度条和标签, 对于那些仍然存在的进度条和标签, 在事件循环还有剩余时间时, 安排它们将自己删除, 然后清除包含它们的容器。如果程序正在终止, 我们就不必这么麻烦了, 因为所有的标签和进度条都在同一个布局中——这意味着, 它们都存在父项, 在删除其父窗口部件 (进度窗口部件) 时, Qt 将用常规的方法将其删除掉。

删除标签和进度条窗口部件, 然后根据需求再创建新的标签和进度条窗口部件, 很明显, 重用而不是重建会比上述做法更有效率。但重用会需要更多的代码, 例如, 隐藏那些已经创建但不再需要的标签和进度条窗口部件 (因为生成的图像比上一次少), 或是创建额外的标签和进度条窗口部件 (因为生成的图像比上一次多)。因此, 如果用户从想要 14 幅图像变成想要 5 幅图像, 就意味着保存了 9 个不需要的标签和进度条。通过删除和创建, 我们减少了处理时间中包含的内存开销。这似乎是一个合理的折中, 因为创建一个新的平滑过渡图像所需要的时间要优于删除标签和进度条并再重新创建它们所需的时间。

通过对 `createAndRunACrossFader()` 方法的调用, 会在 `generateOrCancelImages()` 方法中创建和开始每个 `crossfader` 线程, 我们将分两部分来对此进行介绍。

```
void MainWindow::createAndRunACrossFader(int number,
    const QImage &firstImage, const QImage &lastImage)
{
    QString filename = QString("%1%2.png").arg(baseNameEdit->text())
        .arg(number + 1, 2, 10, QChar('0'));
    QLabel *progressLabel = new QLabel(filename);
    progressLabels << progressLabel;
    QProgressBar *progressBar = new QProgressBar;
    progressBar->setRange(0, 100);
    progressBarForFilename[filename] = progressBar;
    QGridLayout *layout = qobject_cast<QGridLayout*>(
        progressWidget->layout());
    Q_ASSERT(layout);
    layout->addWidget(progressLabel, number, 0);
    layout->addWidget(progressBar, number, 1);
}
```

传入的数字是从 0 开始的, 它是要建立的平滑过渡图像的个数。先从创建图像的合适名称开始 (Image-01.png、Image-02.png, 等等), 可以使用从 1 开始的基于十进制的两位数, 如果需要, 还可以在其最前面再加上一个 0。然后, 向标签列表中添加一个新创建的标签, 还向进度条映射中添加一个新的进度条, 并以文件名作为其键值。提取进度窗口部件的网格布局 (由于前面对 `cleanUp()` 的调用, 网格布局将为空) 将标签和进度条添加到布局中的行, 此行要与图像的数字相对应。

```
double firstWeight = (number + 1) /
    static_cast<double>(numbersSpinBox->value() + 1);
double secondWeight = 1.0 - firstWeight;
CrossFader *crossFader = new CrossFader(filename, firstImage,
    firstWeight, lastImage, secondWeight, this);
crossFaders << crossFader;

connect(crossFader, SIGNAL(progress(int)),
    progressBar, SLOT(setValue(int)));
connect(crossFader, SIGNAL(saving(const QString&)),
    this, SLOT(saving(const QString&)));
connect(crossFader, SIGNAL(saved(bool, const QString&)),
    this, SLOT(saved(bool, const QString&)));
connect(crossFader, SIGNAL(finished()),
    this, SLOT(finished()));
crossFader->start();
}
```

在建立用户接口后,接下来就可以创建并启动用于对图像进行平滑过渡的线程。首先,计算要用到的“窗口部件”。例如,若被创建的图像拥有的比例为(60:40),则其权重就是0.6和0.4。当创建 CrossFader 对象后,需要给定它用来保存图像的文件名,还需给定计算时用到的图像和权重。然后,把 crossfader 添加到 crossfader 列表中,以便可以轻松清除掉它。

由于 CrossFader 是一个 QThread 子类(它自身就是一个 QObject 的子类),我们就可以利用它的那些信号和槽,而不需要创建自定义事件,也不用像我们在前一章的例子中所做的那样去手动调用那些槽。

此处,我们把 crossfader 的自定义 progress() 信号直接连接到进度条上,然后把其他的自定义信号,还有 QThread::finished() 信号连接到相应的主窗口的槽上。最后,调用 QThread::start() 来启动线程的运行。

```
const int StatusTimeout = AQP::MSecPerSecond * 10;
void MainWindow::saving(const QString &filename)
{
    statusBar->showMessage(tr("Saving '%1'").arg(filename),
        StatusTimeout);
    if (QProgressBar *progressBar = progressBarForFilename[filename])
        progressBar->setRange(0, 0);
}
```

无论任何时候,只要 crossfader 完成了一幅图像的创建,在开始保存之前,它都会发射一个自定义的 saving() 信号,这样做的结果就是去调用这个槽。槽会通过状态栏告知用户。它还把对应进度条的范围设定为(0,0),这样一个特殊设定会让进度条显示成“busy”(繁忙)指示器,而不是显示成百分比,这样做很有意思,因为我们不知道保存还需要多长时间才能完成。

```
void MainWindow::saved(bool saved, const QString &filename)
{
    const QString message = saved ? tr("Saved '%1'")
        : tr("Failed to save '%1'");
    statusBar->showMessage(message.arg(filename), StatusTimeout);
    if (QProgressBar *progressBar =
        progressBarForFilename[filename]) {
        progressBar->setRange(0, 1);
        progressBar->setValue(saved ? 1 : 0);
        progressBar->setEnabled(false);
    }
}
```

只要 crossfader 完成了一幅图像的保存,它就发射一个自定义的、导致这个槽调用的 saved() 信号。Boolean 是 QImage::save() 的返回值之一,它表示保存是否成功。与 saving() 方法一样,这种方法

会通过状态栏向用户通知信息。然后,更新相应的进度条,给定它一个任意范围(但这个范围的最大值一定要比最小值大),如果图像保存成功,设定进度条的值为最大值(进度条显示 100%),否则,将其设定为最小值(进度条显示 0%)。进度条也变为不可用,也就不能显示图像处理是否完成的效果。

```
void MainWindow::finished()
{
    foreach (CrossFader *crossFader, crossFaders)
        if (crossFader && !crossFader->isFinished())
            return;
    generateOrCancelButton->setText(tr("Generate"));
    if (canceled)
        statusBar->showMessage(tr("Canceled"), StatusTimeout);
    else {
        statusBar->showMessage(tr("Finished"));
        if (statusBar->checkBox()->isChecked())
            QDesktopServices::openUrl(QUrl::fromLocalFile(
                firstLabel->text()));
    }
}
```

因为有了 saving() 和 saved() 这些信号和槽, crossfader 无论何时结束,它都会发射一个 finished() 信号,此信号会引起这个槽的调用。这个槽会遍历所有的 crossfader,如果发现有尚未完成的 crossfader,它就立即返回并不做任何操作,因为任务还要继续进行。

如果所有的 crossfader 都结束了,Generate 按钮的文本就会从 Cancel 变回来,并通过状态栏通知用户。如果图片是生成结束而不是被取消的,且 Show Images 复选框处于选中状态,就用第一幅图像文件的文件名调用 QDesktopServices::openUrl() 方法,用 QUrl::fromLocalFile() 静态方法将该文件名作为 file:// 协议的 URL 传递过来^①。如果给予 openUrl() 方法的是一个 http:// 协议的 URL,它将试图启动系统的网络浏览器(如果浏览器已经运行,则打开一个新的标签页)。但如果给予 openUrl() 方法的是一个 file:// 协议的 URL,它将启动一个与平台相关的和文件后缀名关联的应用程序(如果这种关联存在的话)。在此情况下,如果计算机有合适的图片浏览器程序,就将启动此程序并把第一幅图像的文件名传送给它。

一些图像浏览器会显示给定的图像,也会显示同目录下其他图像的缩略图,这可以让这些图像之间的导航变得极其容易。当然,为 Cross Fader 程序添加一个图像浏览工具也很简单。我们把它留做练习之用。

对 statusBar->checkBox() 的调用有些出乎意料。使用的不是已创建的自定义 StatusBarBar(在此未做介绍)中的 QStatusBar, StatusBarBar 有一个 QLabel、一个 QCheckBox 和一个 QHBoxLayout 布局中的 QDialogButtonBox,这就是为什么这三者会显示在一“行”中而不是在行下面的状态栏上的原因。图 8.2 给出了这一效果的截图。

我们已经完成了对应用程序用户接口程序结构的介绍,现在可以把注意力放到 CrossFader 的 QThread 子类上,它的所有任务已经完成了。先从介绍头文件中一些提供上下文环境的类的定义开始。

```
class CrossFader : public QThread
{
    Q_OBJECT
```

^① Qt 4.6 还有额外一个静态方法, QUrl::fromUserInput(), 它带一个字符串,返回一个可以使用 file://、ftp:// 或 http:// 协议的 QUrl,所使用的协议取决于输入的字符串。


```

        int blue = qRound((qBlue(firstPixel) * m_firstWeight) +
                           (qBlue(lastPixel) * m_lastWeight));
        image.setPixel(x, y, qRgb(red, green, blue));
        if ((y % 64) == 0 && m_stopped)
            return;
    }
    if (m_stopped)
        return;
    emit progress(qRound(x / onePercent));
}

```

此方法完成了所有的任务。首先,使用 32 位 RGB(红、绿、蓝)色彩创建一个具有正确尺寸的新的 QImage[本可以考虑一个简单的 alpha 通道(透明度)但这恐怕会让代码变得很长,在不改变代码基本结构的情况下,可以把它留做练习]。图像一旦创建完成,就发射一个初始 progress() 信号。这个信号连接到的槽是与众不同的线程——主线程(图形用户界面, GUI)。Qt 处理线程间信号的方式是把它们变成添加到主线程事件序列中的事件。当主线程到达事件时,它会通过调用信号连接的槽进行响应,这样槽就可以在主线程中运行了,而不是在发射信号的辅助线程中运行。

平滑过渡完成的步骤是,通过读取第一幅和最后一幅图像所有像素的 RGB 值,然后为每个像素创建一个新的 RGB 值,此新值是第一幅和最后一幅图像的 RGB 加权值之和,最后再取整。例如,第一幅图像的 R 值为 240,加权值为 0.6,第二幅图像的 R 值为 120,加权值为 0.4,平滑过渡后像素的 R 值将会是 $192:(240 \times 0.6) + (120 \times 0.4) = 144 + 48 = 192$ 。

QImage::pixel() 方法会返回无符号整数;QRgb 只是一个 typedef,它能让 QRgb 的含义更为明确。给 qRed()、qGreen() 和 qBlue() 函数一个 QRgb 值,它们将都会返回一个整数值——一个合适的红、绿或蓝的分量值。QRgba typedef 还包括一个 alpha(透明度)分量和一个 qAlpha() 函数,还有一个 qGray() 函数,给它一个 QRgb 值(或给三个用于红、绿和蓝的整数值),它将会生成一个灰度值。

外部循环沿 x 坐标(列)运行,内部循环沿 y 坐标(行)运行。为了让线程对取消操作敏感,可对每一个第 64 位像素进行检查,以确定线程是否已停止。在每一列完成后还要进行检查并在此信号到达该列后发射一个 progress() 信号。

遗憾的是,使用 QImage::pixel() 和 QImage::setPixel() 非常慢。另外还有两种可选方法,QImage 会把像素数据作为单一连续数组值存储起来,因而这两种方法都是建立在对此情况有所了解的基础之上的。第一种方法是按照水平扫描线(y 轴)工作。QImage::scanLine() 方法返回一个指向像素数据单一“行”的指针。然后,我们可以直接处理这个数据,而不是对像素的获取和设置,这样能让程序明显地得到提速(在示例的源代码中给出了这一方法,但此处未做介绍)。第二种方法是从值所在的数组方面做工作。用 QImage::bits() 方法访问这些数组,然后产生出可能是速度最快的访问方式。此处代码与前面的相同,但这里就使用了 QImage::bits():

```

void CrossFader::run()
{
    QImage image(m_first.width(), m_first.height(),
                 QImage::Format_RGB32);
    emit progress(0);

    const int onePercent = qRound(image.width() * image.height() /
                                   100.0);

    QRgb *firstPixels = reinterpret_cast<QRgb*>(m_first.bits());
    QRgb *lastPixels = reinterpret_cast<QRgb*>(m_last.bits());
    QRgb *pixels = reinterpret_cast<QRgb*>(image.bits()); // Fastest
    for (int i = 0; i < image.width() * image.height(); ++i) {
        QRgb firstPixel = firstPixels[i];
        QRgb lastPixel = lastPixels[i];

```

```

int red = qRound((qRed(firstPixel) * m_firstWeight) +
                 (qRed(lastPixel) * m_lastWeight));
int green = qRound((qGreen(firstPixel) * m_firstWeight) +
                   (qGreen(lastPixel) * m_lastWeight));
int blue = qRound((qBlue(firstPixel) * m_firstWeight) +
                  (qBlue(lastPixel) * m_lastWeight));
pixels[i] = qRgb(red, green, blue);
if ((i % onePercent) == 0) {
    if (m_stopped)
        return;
    emit progress(i / onePercent);
}
}

```

调用 `QImage::bits()` 三次, 这样可以对所有三幅图像进行读写访问。然后, 在一个线性扫描中遍历所有字节, 而不是沿 x 或 y 坐标轴进行迭代(源代码中含有 `#define`, 它在不同方法之间进行转换非常有用——也值得用来做些实验, 以检测使用 `QImage::bits()` 到底有多快)。使这个实现过程变慢的唯一原因是 `if` 语句, 没有它的话, 程序会运行得更快, 但如果用户停止, 会付出更高的代价, 它对于进度的表示也很粗糙。

```

emit progress(image.width());
if (m_stopped)
    return;
emit saving(m_filename);
if (m_stopped)
    return;
emit saved(image.save(m_filename), m_filename);
}

```

只要新的图像一完成, 就可以最终发射一个 `progress()` 信号, 而如果用户没有取消, 就可以发射一个 `saving()` 信号。最后会试图保存图像, 并根据调用 `QImage::save()` 的结果来发射一个 `saved()` 信号。我们并没有明确地发射过 `finished()` 信号; 因为, 一旦 `run()` 结束, 基类会自动替我们完成这项工作。

这个应用程序中使用的 `QThread` 不太难, 因为它不需要锁定, 使我们可以充分利用 Qt 的信号和槽机制。但是, 如果想使并行生成的图像数量比较大, 可以使用工作序列(`work queue`)和一些线程的方式(下一节将介绍这种方法), 它可为你提供一种更具弹性的解决方案。

8.2 共享项的处理

如果需要处理的项比较多, 而且不知道项的大小, 如何在线程间合理地分配这些项以获得最大的输出效果, 使用共享的工作序列或许能较好地改善这一情况。在引入共享工作序列的情况下, 可以先将工作放到序列中, 启动一定数量的辅助线程先来开始数据处理, 然后再逐步添加其他工作。如果工作序列是线程安全的(也就是说, 如果内置了锁定), 那么每一个访问序列的辅助线程都可以将它看做是任何其他类型的数据结构。

本节我们要介绍的情况要稍微复杂一些, 因为我们会先在第一个位置处使用辅助线程填充一个共享数据结构(一个哈希表), 一旦填充完成, 应用程序将填充一个可以在视图中反映哈希表数据的模型。

本节中, 我们将开发一个 Find Duplicates 应用程序(`findduplicates`), 如图 8.4 所示。此应用程序可在用户给定的目录和所包含的子目录内查找重复的文件^①。这里所谓的重复, 是通过考

^① 顺便说一下, 正如前一节讨论的 `Image2Image` 程序一样, 行编辑用来进入到用户选择的根目录, 它使用的是 `QCompleter`, 它会弹出一个有效目录列表, 以减少用户必须键入的内容——这部分内容将在第 9 章中介绍。

察每个文件的大小和 MD5 签名来确定的。MD5 (Message-Digest algorithm 5, 信息摘要算法第 5 版) 是一个密码系统函数, 给它提供一个数据块后 (如一个文件), 它可以提供一个 128 位 (16 字节) 的哈希值。如果两个文件拥有同样的长度和相同的 MD5 值, 那么它们很有可能就包含有相同的内容。

应用程序会遍历所有目录并填充 `QHash < QPair < QByteArray, quint64 >, QStringList >`。哈希表的键值由 `QByteArray` 和 `quint64` 组成, 前者存储 MD5 值, 后者保存文件的大小; 它的值是包含这些文件名的 `QStringList` (包括全路径), 这些文件都拥有相应的 MD5 和大小。一旦哈希表填充完, 就可以遍历它, 进而可以知道那些包括重复文件列表的项, 该项的值会包括有多个文件名。使用这种方法的优点之一是我们甚至可以找到那些文件名不同的相同文件 (例如, 可以参见图 8.4 中 `w39MLRes.DLL` 条目)。

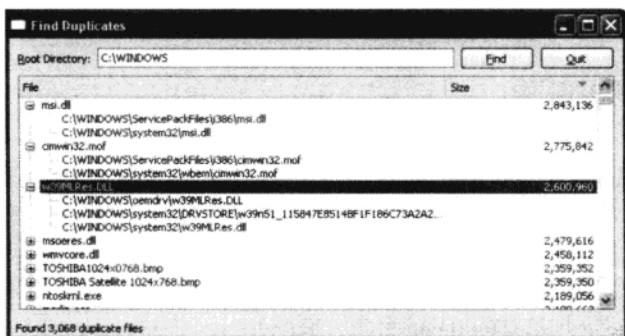


图 8.4 Find Duplicates 应用程序

计算文件的 MD5 值的开销是比较大的 (这与文件的大小成正比), 因此, 我们想把工作分配到一个或更多的辅助线程中去 (具体的理想线程数可由 Qt 报告给出)。但为什么要花心思计算各个 MD5 呢? 我们本可以只将文件的内容存储到哈希键的 `QByteArray` 部分, 特别是, 为什么必须通过某种方式阅读文件来计算 MD5? 因为, 如果这样做就可以最终得到一个哈希表, 它的键消耗的内存与目录的内容消耗的内存一样多 (或许有几十、几百兆的字节, 也可能会更多), 但无论显示的文件有多大, 每个 MD5 的 `QByteArray` 却仅仅只有 16 个字节。

应用程序需要知道它必须处理的文件 (也就是说, 需要计算哪个文件的 MD5 并将其添加到共享哈希表中)。我们采取的办法是, 在用户选择的目录中创建一个子目录列表, 将这个列表尽可能平均地分配到辅助线程中。每一个辅助线程都会向哈希表中添加条目, 这些条目是它所处理的每个目录中的每个文件, 还有每个子目录中的文件。这就无法保证公平分配工作, 例如, 一个辅助线程可能最后分配到的是包含很小图标文件的目录, 而另外的辅助线程分配到的目录中则可能是含有音乐或者 DVD 的大文件。

如果要确保工作的公平分配, 一种办法是创建一个单独的数据结构, 它会包含所有的文件。例如, 根据文件的尺寸键和文件名的自身值按大小排列的 `QMap < quint64, QString >`。然后, 必须根据文件尺寸将文件名分配到线程。例如, 有 3 个辅助线程, 20 个需要处理的文件, 必须按照如下方式分配工作 (在映射中使用索引位置, 但在实际中会只使用一个迭代器): 线程#1 [0, 3, 6, 9, 12, 15, 18]; 线程#2 [1, 4, 7, 10, 13, 16, 19]; 线程#3 [2, 5, 8, 11, 14, 17]。这样做将会稍稍增加程序的复杂度, 但使用这种方法会引起的真正问题是, 或许会比前面那种潜在的不公平分配方式花费的时间要多, 并会毫无疑问地消耗更多的内存。这是因为, 在第一处先创建所有文件的列表时所引入的开销 (即使将这项工作分配给一个或多个辅助线程), 与每个辅助线程只负责处理一个目录相

比,这种方法也不会需要一个太大的文件列表。实际中,对这两种方法进行比较或许比较有意思;至于如何创建一个真正公平的版本及其运行表现如何,可留做练习。

很明显,这种方法的关键在于拥有一个所有辅助线程能够更新的共享哈希表。建立一个百分百线程安全的哈希表不是一件简单的事情,因而我们所做的仅是去创建一个简化的哈希表,它提供了程序所需的功能,但会忽略很多有用但并不需要的功能。

首先从介绍程序的基本要素 ThreadSafeHash 类开始。然后,介绍一些主窗口的基本架构,最后会介绍开始数据处理的 QThread 子类。

以下是 ThreadSafeHash 类的定义,我们一会要介绍的是它的一些方法。

```
template<typename Key, typename Value>
class ThreadSafeHash
{
public:
    explicit ThreadSafeHash() {}
    ...
private:
    mutable QReadWriteLock lock;
    QMultiHash<Key, Value> hash;
};
```

QMultiHash 和 QHash(QMultiMap 和 QMap 与此类似)的不同之处在于,它可以在一个给定键中插入多个值。其结果是,相当于每个键都有了一个多值的列表。与拥有单值的哈希表和映射相比,这样做将导致语义上的细微差别,因此在使用 QMultiHash 或者 QMultiMap 时,必须谨记这些差异的存在(实际上,QHash 和 QMap 都有 insertMulti() 方法,此方法允许把多个值存储到相同的键中,但如非必要,用 QMultiHash 和 QMultiMap 会更方便些)。

我们用到了 QReadWriteLock,因此可以减少锁定时间——例如,如果没有启用写入锁,任何数量的线程都可以获得只读锁。因为要在一些 const 方法中使用该锁,所以它必须是可变的(mutable)。

```
QList<Value> values(const Key &key) const
{
    QReadLocker locker(&lock);
    return hash.values(key);
}
```

此方法会返回与给定键相对应的值,如果哈希表没有包含给定键的项,则返回一个值的空列表。

QReadLocker 会阻塞程序,直到在 QReadWriteLock 上得到一个只读锁,QReadLocker 会作为参数传递给 QReadWriteLock。当 QReadLocker 销毁时,它的析构函数会释放该锁。这就保证了锁总是处于释放状态——无论包含只读锁的函数或方法是正常返回,还是因为出现未处理异常而导致的退出。

对于 Find Duplicates 程序,键的类型是 QPair<QByteArray, qint64>,值的类型是 QString。因此,这个方法返回的值将是一个(有可能为空的)QList<QString>。

QList<QString> 与 QStringList 兼容(QStringList 派生自 QList),但没有提供一些 QStringList 的便捷方法。如果需要的话,可以把 QList<QString> 转换成 QStringList,以便获得更多方法,因为 QStringList 有一个可以接受 QList<QString> 的构造函数。

ThreadSafeHash 有数个其他方法,它们的结构与这个方法的结构相同。也就是说,它们也使用 QReadLocker 并返回对集合哈希表(aggregated hash)的调用结果。这些方法有 contains()、count() 和 isEmpty() (在此未对它们进行介绍)。

```
void insert(const Key &key, const Value &value)
{
    QWriteLocker locker(&lock);
    hash.insert(key, value);
}
```


这个方法用指定的键向哈希表中插入一个单独的值。如果有两个或更多的值插入到相同的键中,它们都会得到保留(按插入顺序)。如果该键不在哈希表中,就会用给定的键和值创建一个新的项。

QWriteLocker 会阻塞程序,直到在 QReadWriteLock 上得到一个可写锁,QWriteLocker 会作为参数传递给 QReadWriteLock。当 QWriteLocker 销毁时,它的析构函数会释放锁——这与 QReadLocker 的处理方式完全一样。

```
const QList<Value> takeOne(bool *more)
{
    Q_ASSERT(more);
    QWriteLocker locker(&lock);
    typename QMultiHash<Key, Value>::const_iterator i =
        hash.constBegin();
    if (i == hash.constEnd()) {
        *more = false;
        return QList<Value>();
    }
    *more = true;
    const QList<Value> values = hash.values(i.key());
    hash.remove(i.key());
    return values;
}
```

提供一个线程安全的迭代机制不是件简单的事,所以我们选择另外一种实现方法来避免那样做,这会破坏性地从哈希表中移除一些任意项,因为这足以满足 Find Duplicates 程序的需求。

由于打算要改变哈希表,故而先从可写锁的获取开始。由于我们头脑中还没有一个特定的键(只是想带一个任意键),所以必须以一定的方式来访问项。此处,我们只是从哈希表中检索“第一个”项的 const 迭代器(把“第一个”放到引号中是因为它不像 QMap、QHash 或 QMultiHash,它是没有内在顺序的)。遗憾的是,迭代器的声明会让一些编译器发生混乱,因此还必须使用 typename 来让它的意义更清晰些。

如果该迭代器指向哈希表结尾部分的后面,就可以知道哈希表是空的。在此情况下,设定 more 指针的布尔值为 false,返回一个值的空列表。

如果哈希表不为空,设定 more 的值为 true,把迭代器的键指向项的检索值的列表(不能使用 i.value(),因为这样将返回一个单独的值,也就是说,只是所有项值列表中的第一个值)。一旦拥有了这些值的备份,就可以从哈希表中移除相应的项并随后返回这些值。在后面将会进一步看到,一旦所有的辅助线程完成了对线程安全的哈希表填充,Find Duplicates 应用程序就会使用这种方法填充模型。

至此,我们已经完成了所有与 ThreadSafeHash 相关内容的介绍。这种数据结构意味着程序中其他部分所用的线程可以有效地把 ThreadSafeHash 当做一个正规的数据结构,可让我们不必再去担心锁定的事情。

现在,我们把注意力放到程序的主窗口结构上,先从介绍它的数据成员开始(但不包括那些标准模型和窗口部件)。

```
volatile bool stopped;
QList<QPointer<GetMD5sThread> > threads;
FilesForMD5 filesForMD5;
```

stopped 变量用来说明用户已取消的那些线程。在后面介绍 processDirectories()、finished() 和 stopThreads() 方法时,将进一步看到,持有辅助线程的列表是很方便的。在本节的最后部分,将介绍 QThread 的子类 GetMD5sThread。filesForMD5 变量是共享的线程安全的哈希表,此哈希表是由辅助线程填充的。

```
typedef ThreadSafeHash<QPair<QByteArray, qint64>,
                      QString> FilesForMD5;
```

我们已经创建了这个 typedef,从而可以方便地在所需的地方轻松键入所有哈希表的名字^①。QPair 用做哈希表的键,QString 是值类型,这些值存储在每一项的值的列表中。

```
void MainWindow::find()
{
    stopThreads();

    rootDirectoryEdit->setEnabled(false);
    view->setSortingEnabled(false);
    model->clear();
    model->setColumnCount(2);
    model->setHorizontalHeaderLabels(QStringList() << tr("File")
                                     << tr("Size"));

    findButton->hide();
    cancelButton->show();
    cancelButton->setEnabled(true);
    cancelButton->setFocus();

    stopped = false;
    prepareToProcess();
}
```

当用户点击了 Find 按钮,就会调用这个方法。它首先停止任何运行中的辅助线程,然后更新用户接口,清除模型,这样就不会显示那些重复的文件了。最好在对一个模型做出大的改变之前,关闭查看排序,这样将改进程序的运行表现。由于清空一个模型的同时也会把它的行和列数设定为 0,也会去掉它的头部,所以为重填模型做准备,我们必须重新存储这些内容。最后,因为用户还没有点击取消,该方法会把 stopped 变量设定为 false,并且会调用 prepareToProcess() 以获取要处理的目录的列表。

此处附带说明的是,没有为 Find 和 Cancel 使用单一按钮并改变其文字,而是使用了两个独立的按钮,在 QHBoxLayout 中依次添加它们。任何时候,两者只有一个可见。这就意味着我们拥有了不必改变按钮文字的方便,也可以使用两个独立的槽 find() 和 cancel(),而不是使用一个单独的 findOrCancel() 槽。

```
const int StopWait = 100;
void MainWindow::stopThreads()
{
    stopped = true;
    while (threads.count()) {
        QMutableListIterator<QPointer<GetMD5sThread> > i(threads);
        while (i.hasNext()) {
            QPointer<GetMD5sThread> thread = i.next();
            if (thread) {
                if (thread->wait(StopWait)) {
                    delete thread;
                    i.remove();
                }
            }
            else
                i.remove();
        }
    }
    Q_ASSERT(threads.isEmpty());
}
```

^① 这里对 typedef 的用法在本书中都较为罕见。实际上,要尽可能多地避免使用 typedef——仅仅是让用户不必在每次碰到它时,都要记住或回顾前面的内容。然而,在产品代码中我们则一定会提倡它的使用。

此方法与前面 Cross Fader 示例中 `cleanUp()` 方法使用了同样的算法,设计用于让总停止时间接近于最慢停止线程的停止时间。

```
void MainWindow::prepareToProcess()
{
    statusBar()->showMessage(tr("Reading files..."));
    QStringList directories;
    directories << rootDirectoryEdit->text();
    QDirIterator i(directories.first());
    while (!stopped && i.hasNext()) {
        const QString &pathAndFilename = i.next();
        const QFileInfo &info = i.fileInfo();
        if (info.isDir() && !info.isSymLink() &&
            i.fileName() != "." && i.fileName() != "..")
            directories << pathAndFilename;
    }
    if (stopped)
        return;
    processDirectories(directories);
}
```

此方法用来创建一个要处理的目录列表。列表中的第一项是用户进入的目录(根目录“root”),其他项(如果有的话)是根目录的子目录。即便对一个很大的目录,创建这个列表也会很快,并且不会消耗太多的内存,因为我们只是深入了一层。一旦列表完成,就可以调用 `processDirectories()` 来完成工作。

如果目录很大,就值得把 `if (stopped)` 检查放到循环中,这样做至少可以保证用户在改变主意的时候可以取消。我们也可以使用多个线程对它进行处理。

```
void MainWindow::processDirectories(const QStringList &directories)
{
    const QVector<int> sizes = AQP::chunkSizes(directories.count(),
        QThread::idealThreadCount());
    int offset = 0;
    foreach (const int chunkSize, sizes) {
        QPointer<GetMD5sThread> thread = QPointer<GetMD5sThread>(
            new GetMD5sThread(&stopped, directories.first(),
                directories.mid(offset, chunkSize),
                &filesForMD5));
        threads << thread;
        connect(thread, SIGNAL(readOneFile()),
            this, SLOT(readOneFile()));
        connect(thread, SIGNAL(finished()), this, SLOT(finished()));
        thread->start();
        offset += chunkSize;
    }
}
```

此方法分配工作的算法与前一章的 `convertFiles()` 方法相同。每一个 `GetMD5sThread` 对象都会根据指向 `stopped` 变量的指针(以便在用户取消时它可以检测到)、根目录(因为这个目录不能递归进入)、要处理的目录和线程要更新的哈希表进行创建。

`GetMD5sThread` 一旦创建,就把它添加到线程表中。然后,将线程的自定义 `readOneFile()` 信号和继承过来的 `finished()` 信号连接到主窗口中的相应槽,启动线程进行处理。

```
void MainWindow::readOneFile()
{
    statusBar()->showMessage(tr("Read %Ln file(s)", "",
        filesForMD5.count()));
}
```

无论何时,只要 `GetMD5sThread` 向哈希表中添加一个文件名,它都会发射一个 `readOneFile()` 信号,

此信号会转而调用这个槽。既然 FilesForMD5 哈希表是线程安全的,我们就没有必要担心对其计数值的锁定访问问题了;当然,该计数值是所有辅助线程的总和,因为它们会共享同样的哈希表。

严格地说,这个计数值是错误的,因为它度量的是唯一的 (MD5, size) 值对,而不是实际处理的文件个数——此数字可能会大一些,这取决于存在的重复文件的个数。尽管如此,这个计数值还是充分地接近了显示处理进度的正确值,这很好地说明了某些时候在精确度和效率之间的一种折中方法。此处,我们不需要精确度(显示的数字总是要持续变化,让用户知道数据处理正在进行就可以了)。因此,我们没有必要去尝试计算真正的结果,这样会浪费 CPU 周期或内存空间。

```
void MainWindow::finished()
{
    foreach (QPointer<GetMD5sThread> thread, threads)
        if (thread && thread->isRunning())
            return;
    processResults();
}
```

无论何时,只要 GetMD5sThread 一结束,它就发射一个 finished() 信号(从它的 QThread 基类继承的动作行为)。因为可能存在多个线程,检查所有这些线程,以确定是否仍然有线程在运行,如果有,则返回,不做任何处理;如果没有,辅助线程仍在运行,我们就可以知道,最后结束的线程调用了这个槽来结束运行,这样就能调用 processResults() 了。

```
void MainWindow::processResults()
{
    stopThreads();
    qint64 maximumSize;
    forever {
        bool more;
        QStringList files = filesForMD5.takeOne(&more);
        if (!more)
            break;
        if (files.count() < 2)
            continue;
        addOneResult(files, &maximumSize);
    }
    updateView(maximumSize);
    statusBar()->showMessage(tr("Found %Ln duplicate file(s)", "",
                                model->rowCount()));
    completed();
}
```

这个方法用来向用户显示结果。它首先调用一个删除所有辅助线程的 stopThreads(), 因为我们不再需要这些线程了。

需要追踪最大尺寸的文件,因为这可以决定视图的 Size 列的宽度。ThreadSafeHash 基本框架没有为索引项提供迭代器或任何方法,除了它们的键(也没有给出提供键的任何方法);它有的只是一个破坏性的 takeOne() 方法,但这已足以满足我们的要求了。使用 Qt 的 forever 宏(实际上与 while(1) 相同)来启动一个无限循环来尝试检索文件名列表(ThreadSafeHash::takeOne() 方法会返回一个 QList<QString>; 依靠非显式构造函数 QStringList(QList<QString> &) 来实现转换)。如果 more 设定为 false, 哈希表为空,就跳出循环。否则,提供至少包括两个项的文件名列表(也就是说,至少有一个重复文件),并利用这一信息调用 addOneResult() 来填充该模型。

最后,更新视图(对其进行分类并设置它的列宽),告诉用户有多少个重复文件,调用 completed(), 为另一个查找准备用户接口。

```
void MainWindow::addOneResult(const QStringList &files,
                             quint64 *maximumSize)
{
    QFileInfo info(files.first());
    if (info.size() > *maximumSize)
        *maximumSize = info.size();
    QStandardItem *parentItem = model->invisibleRootItem();
    QStandardItem *treeItem = new QStandardItem(info.fileName());
    QStandardItem *sizeItem = new QStandardItem(
        QString("%L1").arg(info.size(), 20, 10, QChar(' ')));
    sizeItem->setTextAlignment(Qt::AlignVCenter|Qt::AlignRight);
    parentItem->appendRow(QList<QStandardItem*>() << treeItem
                        << sizeItem);

    foreach (const QString &filename, files)
        treeItem->appendRow(new QStandardItem(
            QDir::toNativeSeparators(filename)));
}
```

这个方法会基于文件名列表中的第一个文件名创建一个新的顶层行。行的第一项(也就是它的第一列)是给定的文件名(从它的路径中提取出来),行的第二项(也就是它的第二列)是给定的文件尺寸(是一个字符串)。然后,该方法在列表中为每个文件名都添加一个子行,为每行创建一个项来保存文件名(包括它的路径)。这意味着,每一个顶级行至少有两个子行。同时要注意的是,子项可能不与顶层行的第一项同名,因为 Find Duplicates 查找重复文件时不会考虑实际的文件名。

文件尺寸会以本地化的字符串形式显示出来,但会留前导空白,这样的话,如果用户对尺寸列进行排序(通过点击 Size 列的表头),将默认按字母次序对该值进行排序。当然,使用 QDir::toNativeSeparators() 静态方法可保证正确地显示与系统平台相关的路径分隔符。

```
void MainWindow::updateView(quint64 maximumSize)
{
    if (model->rowCount()) {
        model->sort(0, Qt::AscendingOrder);
        view->expand(model->invisibleRootItem()->child(0)->index());
        QFontMetrics fm(font());
        int sizeWidth = fm.width(QString("%L1W").arg(maximumSize));
        view->setColumnWidth(1, sizeWidth);
        sizeWidth += fm.width("W");
        view->setColumnWidth(0, view->width() - (sizeWidth +
            view->verticalScrollBar()->sizeHint().width()));
    }
}
```

一旦用重复文件填充完模型就会调用这个方法。它通过顶层文件名对模型进行排序并通过扩展第一项来显示它的重复文件。然后,为尺寸列计算一个合适的宽度,使用本地化的最大尺寸和两个“W”来提供一些水平方向的边距。最后,用剩余值设定文件名列表的宽度,这还要考虑垂直滚动条加上一些额外边距的宽度。

```
void MainWindow::completed()
{
    view->setSortingEnabled(true);
    cancelButton->setEnabled(false);
    cancelButton->hide();
    findButton->show();
    findButton->setEnabled(true);
    findButton->setFocus();
    rootDirectoryEdit->setEnabled(true);
}
```

一旦查找完成(或取消)就会调用这个方法来准备下一次查找所需的用户接口。这个方法还重新启用了排序功能(因为对模型的所有改变都已完成),以使用户通过点击列表头,就可以对文件进行排序了。

```
void MainWindow::cancel()
{
    stopThreads();
    completed();
    statusBar()->showMessage(tr("Canceled"), StatusTimeout);
}
```

如果用户取消,则调用这个槽。它会停止所有线程,为下一次查找准备所需的用户接口,并确认取消操作。它没有执行的任务是清除模型,所以,目前为止找到的任何重复项都会显示出来。当然,要改变这一点也很容易。

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    stopThreads();
    event->accept();
}
```

如果用户试图终止程序,就需要首先确定所有的辅助线程都已结束,然后再允许终止操作的继续进行。

目前为止,关于程序结构的内容我们已经介绍的足够多了,看到了是如何创建线程、如何连接线程和使用线程。现在将介绍(简单得出乎意料的)GetMD5sThread类。首先看一段从它的定义中提取的代码,然后会介绍它的run()方法。

```
class GetMD5sThread : public QThread
{
    Q_OBJECT
public:
    explicit GetMD5sThread(volatile bool *stopped,
        const QString &root, const QStringList &directories,
        FilesForMD5 *filesForMD5)
        : m_stopped(stopped), m_root(root),
          m_directories(directories), m_filesForMD5(filesForMD5) {}
signals:
    void readOneFile();
private:
    void run();
    ...
};
```

构造函数带有用来定义线程必须执行的工作的一些参数,它会将执行的工作存储到成员变量中。只声明了一个信号,因为finished()信号是从QThread继承的。通过将它的run()方法私有化,就禁止了类的子类化,但也禁止对实例进行的run()调用(因为run()仅能被QThread::start()所调用)。

```
void GetMD5sThread::run()
{
    foreach (const QString &directory, m_directories) {
        QDirIterator::IteratorFlag flag = directory == m_root
            ? QDirIterator::NoIteratorFlags
            : QDirIterator::Subdirectories;
        QDirIterator i(directory, flag);
        while (i.hasNext()) {
            const QString &filename = i.next();
            const QFileInfo &info = i.fileInfo();
            if (!info.isFile() || info.isSymLink() ||
                info.size() == 0)
                continue;
            if (*m_stopped)
                return;
            QFile file(filename);
            if (!file.open(QIODevice::ReadOnly))
                continue;
        }
    }
```

```

        QByteArray md5 = QCryptographicHash::hash(file.readAll(),
            QCryptographicHash::Md5);
        if (*m_stopped)
            return;
        m_filesForMD5->insert(qMakePair(md5, info.size()),
            filename);
        emit readOneFile();
    }
}
}

```

这个方法是这个应用程序的核心所在,也是所有工作完成的地方。线程遍历给定的目录列表,并为每个列表都创建一个 QDirIterator,用它来遍历所有将被处理的文件。原始目录列表由用户所选择的目录(根目录“root”)和根目录的直接子目录所填充。这意味着,第一个线程的第一个目录将是一个根目录,这个根目录已经处于将要处理的列表中。因此,如果将要处理的目录是根目录的话,我们将不能再进入它的子目录——这是需要通过设定 QDirIterator 标志来加以考虑的东西。

我们忽略了所有不是文件的内容,还忽略了 0 长度的文件(即便在逻辑上,这些是其他文件的重复)。对于那些我们想要处理的文件,以二进制模式打开它们,让程序进行读取,如果调用 QFile::open() 成功,将它们的所有内容传递到 QCryptographicHash::hash() 函数,以便使用 QFile::readAll() 来计算它的 MD5 值。然后更新哈希表数据结构,创建(或读取)一个项,此项的键含有所处理文件的 MD5 值和文件尺寸,把文件名添加到项的字符串值的列表中。Qt 的全局函数 qMakePair() 用来创建 QPair 对象——在这个示例中是一个(QByteArray, quint64)键。当然,我们不必担心锁定问题,因为 ThreadSafeHash 已经为我们考虑到了这一点。

最后,发射一个说明文件已被读取的信号。QThread 是 QObject 的子类,可以正常地使用 Qt 的信号和槽机制——不必借助于自定义事件或是直接对槽的调用(在底层,Qt 使用事件处理机制来处理线程间的信号-连接,但这些都是以不可见的方式完成的,因此我们没有必要知道或是关心它是如何工作的)。

值得注意的是,有好几个地方我们会检查用户是否取消了操作,如果是,则立即结束。

在 Qt 4.3 中引入了 Qt 的 QCryptographicHash 类。它可以使用任意的 MD5、MD4 或是 SHA1^① 算法提供加密哈希表。此处使用的静态方法 QCryptographicHash::hash() 带一个 QByteArray (QFile::readAll() 的返回内容) 参数和所要使用的算法,可返回一个 QByteArray 加密哈希表。创建一个 QCryptographicHash 对象并使用它的 addData() (带一个 char * 和一个长度或者带一个 QByteArray 参数) 方法中的一种来向它添加数据也是有可能的,最后,为哈希表调用 QCryptographicHash::result()。

至此,我们已经介绍完了 Find Duplicates 应用程序。这个应用程序说明了如何让 QThread 与一个线程安全的数据结构一起使用。一个稍微更为有用的变化是,用第三列显示每一个重复文件的数量,这样用户可以通过此列来排序,以查看哪个文件重复得最多,而不必去关注文件名和尺寸大小。另一个改进或许应该是删除文件,对于类 UNIX 操作系统来说,则是连接文件的功能(也就是说,删除重复的文件,然后从一个文件版本向当前删除的重复版本之间建立一个软件链接)。我们把添加这些功能留做练习。

现在,我们已经完成了 Qt 的高级线程处理类(在前一章)和 QThread 的介绍。Qt 对于线程处理的支持是相当出色的,但一个不可避免的现实是,编写(特别是维护)拥有线程处理功能的

① 即 Secure Hash Algorithm1,安全哈希算法第1版——译者注。

程序,要比单线程程序难得多。有鉴于此,线程应当仅在需要的时候才去使用它,并且务必要小心谨慎。我们可以通过把线程处理压缩到一个类中来最小化风险,最大化收益,它可以完全避免锁定问题(通过项的独立处理)或是在需要锁定的地方,这样类的客户端就不必自己担负任何锁定的责任。使用 Qt 的 QtConcurrent 函数和 QRunnable 类,可以将独立项的处理变得非常简单——如果需要的话(例如,在 7.2 节中创建的 ThreadSafeErrorInfo 类),我们仍然可以使用锁定。使用 QThread 类的需求条件不是非常苛刻,特别是在当我们使用线程安全的数据结构的话,虽然编写这些将是一个非常大的挑战,但如果想要它们变得更加高效并且要实现全部功能的话,还是很值得一试的。



第9章 创建富文本编辑器

- QTextDocument 简介
- 一个单行的富文本编辑器
- 创建自定义的文本编辑器
- 编辑多行的富文本

Qt 提供了一个富文本引擎,我们可以用它来格式化并显示文本、列表、图表和图像^①。这一文本格式化引擎的核心是 QTextDocument——这个类可以对一个单独的文本片段、一行或是整个多页面文档,甚至单个字符进行格式化,它完全支持文本格式化(例如,粗体、斜体、颜色、下标)。

使用 QTextDocument 的最大便捷之处在于它能接受 HTML,这使得在程序中引入富文本非常容易。正如 Web 浏览器一样, QTextDocument 可以接受 CSS(层叠样式表, Cascading Style Sheet)为它所包含的文本提供一种全局一致的格式。 QTextDocument 的另一个与 Web 浏览器相同的属性是它可以安全地忽略不能理解的标记。 Qt 的富文本引擎支持的 HTML 标签和 CSS 属性已在 qt.nokia.com/doc/richtext-html-subset.html 中列出。

本章中,我们将把重点放在创建富文本编辑器上,包括输入提示和语法加亮。下一章将把重点放到富文本的输出上,包括输出到文件。例如,到 .odt(开放文档文本格式, Open Document Text Format) 和 .pdf(便携文档格式, Portable Document Format) 和打印。

在 9.1 节中,先从浏览 QTextDocument 类开始,它能够富文本文档提供常驻内存存储的方法。这部分简介内容与本章以及下一章的内容相关。然后,在 9.2 节中,将介绍如何为一个行编辑器提供输入提示,然后创建一个自定义的多行 XML 编辑器,并为其提供输入提示和语法加亮功能。9.3 节将介绍如何创建一个单行的富文本编辑器。9.4 节将介绍如何创建一个多行的富文本编辑器——它允许我们提供比单行编辑器更多的特性,如文本对齐、字符提示,以及光标所在位置的段落格式化属性。

9.1 QTextDocument 简介

这一节中,我们将简要描述 QTextDocument 的结构。使用 QTextDocument 来操控自定义编辑器的富文本,在本章和下一章中,将以自定义的编程方法使用 QTextCursor 对 QTextDocument 进行编辑,并对它进行填充。

富文本的存储似乎是件非常简单的事情,但即使粗略地看一下那足有 700 页的开放文档格式(Open Document Format)说明文件(或者是微软公司 6000 多页的 OOXML 格式说明文件),也足以让我们知道这件事情或许并不像我们想象的那样简单。庆幸的是,Qt 的富文本仅支持一些经过慎重选择的功能,因此学习起来不会太难,而且这些足够满足日常使用了。

QTextDocument 在内部使用一个递归结构来存储它的文档,这个结构的组成包括一个含有一个(可能为空)文本块的根框架,然后是一系列的 0 或者许多框架、文本块或表格。根下的每个框架都包括一个(可能为空)文本块,然后又是一系列的 0 或者更多的框架、文本块或表格。这种格

^① Qt 的富文本格式是一个常驻内存的数据格式,不要与 Microsoft 的 .rtf(富文本格式, rich text format)文件交换格式、富文本(RFC 1896, enriched text)和 MIME 类型(RFC 1341 和 1521)的文本/富文本(text/richtext)格式相混淆,它们是完全不同的。

式在框架中的框架会一直循环下去。Qt 通常把一个文本块(即便为空)当做一个框架或者表格间的分隔符。

图 9.1 给出了一个示例文档第一页的 QTextDocument 的结构图。第二页也在根框架中,它就在图中显示的最后的文本块之后。这两个页面都包括一个文本块(拥有一个标题)、一个文本表格(拥有单元格,每一个单元格包括一个带有标题的文本块)和另一个文本块(在它的末尾处有一个段落,所有的页面都一样,最后一页有所不同,它含有一个页面分隔符)。

QTextBlock 可以代表一个段落,或者一个列表,也完全支持嵌套列表。如果文本块代表一个列表, QTextBlock::textList() 方法返回一个指向 QTextList 的指针;否则,(如果它只是一个文本段落)返回 0。一个 QTextList 由一个或者多个 QTextBlock 组成,这些 QTextBlock 的属性存储在一个单独的 QTextListFormat 中。

QTextBlock 的段落格式设置存储在一个 QTextBlockFormat 中——存储的属性包括段落的对齐、边距、缩进等。0 个或更多的 QTextFragment 组成一个文本块,每一个 QTextFragment 都有

一段文本,这些文本的属性(字体、下画线,等等)存储在一个单独的 QTextCharFormat 中。文本可以像一个字符那样小(如一个带有下画线的字母),也可以像一个文本段落那样大。若一个非空文本块的文本也拥有相同属性,则它通常会包括一个单独的文本段。

不要把 QTextFragment 类和 QTextDocumentFragment 类混为一谈。QTextBlock 是由一个或多个 QTextFragment 组成的,而 QTextDocumentFragment 是用来保存 QTextDocument 的任意片段的,它可以包含段落、表格甚至是一个完整的 QTextDocument。当用户做出选择时,可以用 QTextCursor::selection() 方法(它返回一个 QTextDocumentFragment)来获取用户选择的内容。

图像可以用 QTextFragment 中包含的占位符(Unicode 字符 U + FFFC)来表示。这个占位符有一个用来保存图像尺寸和名称的 QTextImageFormat(QTextCharFormat 子类)。它的名称是应用程序资源中一幅图像的名称。

表格可以使用 QTextTable 类来表示,QTextTable 类的属性(对齐、单元格填充、单元格间距、列数,等等)存储在单独的 QTextTableFormat 中。表格单元格由 QTextTableCell 类表示,它可以包含文本块或者框架;因此可能存在复杂的嵌套。单元格知道自己在表格中的位置(它们所在的列和行),单元格还拥有列和行的宽度属性和一个 QTextCharFormat。

表格和框架在文档结构内会用同样的方式加以处理,因为 QTextTable 类是 QTextFrame 类(用来表示框架的类)的一个子类。每一个框架的属性(边、边外空白、内填充、宽度等)都存储在单独的 QTextFrameFormat 中。

Qt 有很多与富文本相关的类,但其中一些主要关注于对文本进行布局而不是存储文档元素。一个非常重要的与 QTextDocument 相关的类是 QTextCursor,它为编辑 QTextDocument 提供了一种可编程手段。在这一章,我们使用 QTextCursor 来编辑已有的文档,下一章用它从头开始创建文档。

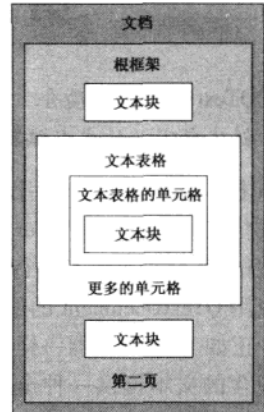


图 9.1 示例文档第一页的示意图

9.2 创建自定义的文本编辑器

这一节中,我们将介绍如何把行编辑和纯文本多行编辑器变得更便于用户使用。

首先,要为他们增加的功能是输入提示(completion)。输入提示是在用户键入文本(或者在一些执

行情况下,键入一个特殊的键序列)时提示可选文本,以完成用户所键入的内容,它通常以一个列表的形式呈现。用户可以通过使用方向键并按下 Enter 来在列表中浏览并选择一个文本,或通过点击所想要的选项。也可以忽略提示列表,通过继续输入,或按下 Esc 或点击列表外部区域^①。

还需要为多行编辑器添加的功能是,当前行的加亮显示和彩色语法加亮。当前行加亮可以让用户更加容易地看到它们所处的位置,彩色语法加亮有助于显示文本的结构(为用户使用的特殊语法而做,本例中为 XML),还有助于用户查找语法错误。

9.2.1 行编辑和组合框的输入提示

在 Image2Image(参见第7章)和 Find Duplicates 应用程序中(参见第8章),我们都使用了 QCompleter 和 QLineEdit,来为用户在输入路径时提供输入提示。在这两个程序中,虽然 QCompleter 也能以行内提示的形式出现,但我们使用的是一个弹出式输入提示;这两种类型如图 9.2 所示。

QCompleter 类获取一个 QAbstractItemModel(它包括一个列表或者是项的树结构),把模型的项用做输入提示选项。QCompleter 使用的模型可能会对自己包含的项排序,也可能不排序。

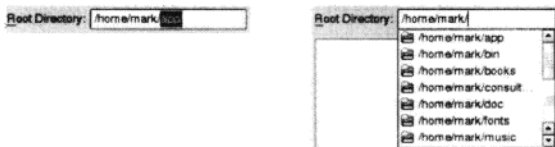


图 9.2 行内和弹出式输入提示

下面是 Find Duplicates 程序的 MainWindow::createWidgets() 方法,此处创建了行编辑和输入提示功能:

```
void MainWindow::createWidgets()
{
    rootDirectoryLabel = new QLabel(tr("Root Directory:"));
    rootDirectoryEdit = new QLineEdit(QDir::toNativeSeparators(
        QDir::homePath()));
    rootDirectoryLabel->setBuddy(rootDirectoryEdit);
    QCompleter *directoryCompleter = new QCompleter(this);
    directoryCompleter->setCaseSensitivity(Qt::CaseInsensitive);
    directoryCompleter->setModel(new DirModel(directoryCompleter));
    rootDirectoryEdit->setCompleter(directoryCompleter);
}
```

提供输入提示需要的所有内容是创建一个提示器,给予它一个模型,并设定它在编辑器窗口部件上显示。因为一旦这些所有的都建立好了,Qt 会自动处理键盘交互,并弹出列表。

在这种特殊情况下,我们不像往常一样使用 QDirModel,而是在其基础上创建一个自定义子类^②。这样,就可以在所有平台上正确地显示路径分隔符了。基于完整性的考虑,下面是自定义模型的代码(它与在 Qt 的 examples/tools/completer 示例中的代码几乎完全一致):

```
class DirModel : public QDirModel
{
public:
    explicit DirModel(QObject *parent=0) : QDirModel(parent) {}
}
```

① 与输入提示非常相近的一个概念是输入法(input method)——用户可以通过它们输入文本。日常生活中就有这样的例子,如通过按下手机上的数字键来输入文本信息。桌面计算机方面的例子则是,一名日本用户键入一定序列的拉丁字符,它们会被程序接收到,这些字符则会用做日文字符;更多详情请参阅类 QInputContext 的技术文档。

② 从 Qt 4.7 开始, QDirModel 可能会被一个异步的 QFileSystemModel 类取代。

```

QVariant data(const QModelIndex &index,
              int role=Qt::DisplayRole) const
{
    if (role == Qt::DisplayRole && index.column() == 0) {
        QString path = QDir::toNativeSeparators(filePath(index));
        if (path.endsWith(QDir::separator()))
            path.chop(1);
        return path;
    }
    return QDirModel::data(index, role);
}
};

```

这个模型子类用 `QAbstractItemModel::data()` 方法取代了 `Qt::DisplayRole`, 以保证为用户显示路径时使用本地分隔符, 并且不是以一个分隔符结尾。

很明显, 为 `QLineEdit` 或 `QComboBox` 建立一个提示器是很简单的。`QComboBox` 也有一个以同样方式工作的 `setCompleter()` 方法。如果有一个字符串的静态列表, 建立过程将非常简单, 因为在此情况下, 可以把这些字符串传递给 `QCompleter` 构造函数, 而完全不需要建立一个模型。

在某种程度上来说, 为一个多行文本编辑器创建输入提示器是极具挑战性的, 因为必须提供一些启用它的方式 (例如, 一个特殊的键序列), 还要填充并为它正确地定位。下一小节将介绍这些工作是如何完成的。

9.2.2 文本编辑器的输入提示和语法加亮

在本小节中, 将为 XML 开发一个基本的文本编辑窗口组件。它除了从 `QPlainTextEdit` 和 `QTextEdit` (如复制和粘贴、撤销/重做和缩放) 免费得到的功能之外, 还有我们将要添加的两个主要功能, 它们是输入提示和语法加亮。

这一节中, 我们将介绍 `XMLEdit` 窗口组件 (xmledit), 如图 9.3 所示。左边的图像是, 用户已经输入了 `des`, 然后按下 `Ctrl + M` 组合键 (我们用来启动输入提示的键盘快捷键) 的效果, 在输入提示模型中只有一个词以 `des` 开头。在此情况下, 编辑器立即插入匹配单词的未输入的部分, 并选择插入的文本。用户可以通过按下 `Enter` 键 (此时, 插入的文本将变成未选择, 光标移动到单词的后面) 接受插入, 也可以通过按下 `Esc` 键 (或动过按下 `Del` 键, 或通过输入更多字母) 来拒绝插入内容。右边的图像是, 用户已经输入 `cat`, 然后按下 `Ctrl + M` 组合键的效果, 在输入提示模型中多个以 `cat` 开头的单词。此时, 编辑器弹出一个供用户选择的单词列表。用户可以从列表中, 通过点击选择一个单词, 或使用上下方向键浏览, 然后在加亮的单词上按下 `Enter` 键来选择。也可以通过按下 `Esc` 再点击其他地方, 而不是列表本身或继续输入内容来取消^①。

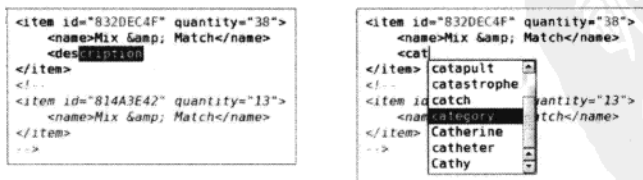


图 9.3 XMLEdit 中选择一个词和多个词的输入提示

① 在大部分平台上都没有为输入提示标准化的按键, 例如, 不同程序在 Linux 和 Windows 下使用自己唯一的按键设定。然而, 在 Mac OS X 上, `Esc` 是为输入提示而标准化的按键, 如果想使用这个标准按键, 我们可以使用 `#ifdef Q_WS_MAC` 来设定 `Esc` 为 Mac OS X 上输入提示的按键。

图中显示的 XML 也给我们留下了颜色语法加亮的初步印象。我们已选择标记为深蓝色、属性名为蓝色、属性值为深黄色、标题为深红色、注释为绿色且斜体。还要注意的,两幅图像都加亮显示了用户所在的行的背景(我们在屏幕截图中使用了比在源代码中更深的加亮颜色;请创建并运行程序,以查看实际使用的颜色)。

如果想更进一步,添加行号也是比较简单的,我们把这留做练习。参考 Qt 的 `examples/widgets/codeeditor` 示例,看它是如何完成的。但如果想要功能全面的 Qt 纯文本编辑器,要让它拥有语法加亮,支持多语言,并且对这些功能完全自主,最好还是使用 QScintilla (www.riverbankcomputing.co.uk/software/qscintilla),它是 C++ Scintilla 编辑器组件 (www.scintilla.org) 的一个 Qt 移植版本。

9.2.2.1 多行编辑器的输入提示

对于 XmlEdit 窗口部件,我们将提供两种输入提示。如果用户启动输入提示,只有一个备选单词,将立即插入这个词的其余部分,插入的文本将处于被选中状态。此时,如果用户按下 Esc 键或是 Del 键,或者键入除了 Enter(接受插入)以外的其他内容,已选择的文本将被移除,在用户输入一个字母时,将像往常一样插入输入的字母。如果用户启动输入提示,且有多个备选项时,将显示一个包括有备选单词的弹出式列表,用户可以在其中选择,或是取消弹出列表。

我们已选择将 QPlainTextEdit,而不是 QTextEdit 进行子类化,把它作为 XmlEdit 类的基础。QPlainTextEdit 类有一个稍微有点误导人的名称(它或许应该被称为 QBasicTextEdit)因为它全面地支持字符格式化,如粗体、斜体和颜色,甚至还提供了 QPlainTextEdit::appendHtml()方法! QPlainTextEdit 与 QTextEdit 非常相似,它支持很多相同的功能,包括使用 QSyntaxHighlighter,下一小节将看到它。两者关键性的差异在于,QPlainTextEdit 不支持框架和表格,它使用更简单的算法来布局文本。这意味着在处理大型文档时,QPlainTextEdit 比 QTextEdit 更快,这样,把它用做日志浏览器是比较理想的,这也使它成了创建自定义文本编辑器的基础。

为了支持输入提示,XmlEdit 类要包含三个私有成员变量:bool 类型的 completedAndSelected、QCompleter * 类型的 completer 和类型为 QStringListModel * 的 model。另外,除了要用构造函数和方法创建窗口部件的子部件,并建立窗口部件的连接,我们还必须重新实现两个事件处理器,并提供 3 个槽和 6 个(小的)支持方法,以完成函数的实现。首先看一下提供上下文的构造函数。

```
XmlEdit::XmlEdit(QWidget *parent)
    : QPlainTextEdit(parent), completedAndSelected(false)
{
    createWidgets();
    createConnections();
    highlightCurrentLine();
}
```

该构造函数相当传统。在下一小节学习语法高亮时再介绍 highlightCurrentLine() 函数。

```
void XmlEdit::createWidgets()
{
    (void) new XmlHighlighter(document());
    model = new QStringListModel(this);
    completer = new QCompleter(this);
    completer->setWidget(this);
    completer->setCompletionMode(QCompleter::PopupCompletion);
    completer->setModel(model);
    completer->setModelSorting(
        QCompleter::CaseInsensitivelySortedModel);
    completer->setCaseSensitivity(Qt::CaseInsensitive);
    completer->setWrapAround(true);
}
```

第一个调用创建了自定义语法加亮器,并将它应用到 `QTextDocument` (来自基类), `QTextDocument` 拥有 `XmlEdit` 的文本和格式化数据。

其他方法是关于输入提示的。首先,为将要使用的输入提示器创建一个模型/视窗模型,然后创建一个输入提示器。我们已选择使用弹出式输入提示模型,但也可以用 `QCompleter::InlineCompletion` 或 `QCompleter::UnfilteredPopupCompletion` 来代替。

调用 `QCompleter::setModelSorting()`,这样就能告诉输入提示器,模型是如何排序的——它不会导致任何排序的完成!我们已说过,模型使用的是大小写敏感的排序方式;其他选项为 `QCompleter::CaseSensitivelySortedModel` 和 `QCompleter::UnsortedModel`。如果输入提示模型列内的和输入提示相关的数据已排序,那么,告诉输入提示器模型已排序(正如此处所完成那样)。如果输入提示器知道输入提示模型已排序,那么在查找输入提示时,它将使用一个快速的二分法搜索,而不是较慢的线性搜索——这样,大部分输入提示模型就会有相当大的性能改进。

调用 `QCompleter::setCaseSensitivity()`,这样就能告诉输入提示器,是否需要大小写敏感——如果不需要,它将显示并插入输入提示,而不考虑大小写,否则,它将显示并插入输入提示,这些输入提示的大小写与将要完成文本的大小写相匹配。最后,调用 `QCompleter::setWrapAround()` 来做出决定,如果用户浏览到输入提示列表的底部或顶部时将做何处理。例如,设定为 `true` 时,当定位到最顶端的(第一个)项“之上”时,将用户带到最底部的(最后一个)项。

```
void XmlEdit::createConnections()
{
    connect(this, SIGNAL(cursorPositionChanged()),
            this, SLOT(highlightCurrentLine()));
    connect(completer, SIGNAL(activated(const QString&)),
            this, SLOT(insertCompletion(const QString&)));
    (void) new QShortcut(QKeySequence(tr("Ctrl+M", "Complete")),
            this, SLOT(performCompletion()));
}
```

此处需要三个信号-槽连接,后两个为输入提示所用。第一个连接用来确保,当光标移动(通过用户键入或浏览,例如,使用方向键或点击)后,加亮当前行。我们将在下一小节讨论此内容。

当用户从输入提示列表中选择一项时,输入提示器发射 `activated()` 信号;连接到此信号,以便在此情况发生时,插入适合的输入提示。

第三个连接用来创建一个新的 `QShortcut` 的一部分,且用来保证当用户按下 `Ctrl + M` 组合键时,调用 `performCompletion()` 槽。

```
void XmlEdit::performCompletion()
{
    QTextCursor cursor = textCursor();
    cursor.select(QTextCursor::WordUnderCursor);
    const QString completionPrefix = cursor.selectedText();
    if (!completionPrefix.isEmpty() &&
        completionPrefix.at(completionPrefix.length() - 1)
            .isLetter())
        performCompletion(completionPrefix);
}
```

当用户键入 `Ctrl + M` 组合键,启动输入提示,如果真的存在需要完成的文本,只要把输入提示过程设为运行即可。这个槽首先从带有下画线的 `QTextDocument` 处获取一个光标,然后选择,并提取光标所在的单词(可以为单独的字符),或者是光标之前紧接的单词(没有中间空白)。如果获取到的光标不是在一个单词中间,或是紧随着一个单词,那么将获取到一个空字符串。

如果单词(也就是输入提示的前缀)不为空,且以一个字母结尾,利用前缀调用 `private` 过载的 `performCompletion()` 方法。

注意,我们没有调用 `QPlainTextEdit::setTextCursor()` 来修改光标。这意味着,我们所做的改变(也就是选择一个单词)没有应用到文档,但此处,想要让它应用到文档。

除了选择单个词的功能以外, `QTextCursor::select()` 方法可以用来选择当前行(`QTextCursor::LineUnderCursor`)、当前段落(`QTextCursor::BlockUnderCursor`),甚至整个文档(`QTextCursor::Document`)。 `QTextCursor` API 在表 9.1、表 9.2 和表 9.3 中显示。

```
void XmlEdit::performCompletion(const QString &completionPrefix)
{
    populateModel(completionPrefix);
    if (completionPrefix != completer->completionPrefix()) {
        completer->setCompletionPrefix(completionPrefix);
        completer->popup()->setCurrentIndex(
            completer->completionModel()->index(0, 0));
    }
    if (completer->completionCount() == 1)
        insertCompletion(completer->currentCompletion(), true);
    else {
        QRect rect = cursorRect();
        rect.setWidth(completer->popup()->sizeHintForColumn(0) +
            completer->popup()->verticalScrollBar()->
            sizeHint().width());
        completer->complete(rect);
    }
}
```

这个方法首先填充输入提示器所使用的模型。然后,就能保证输入提示器使用的输入提示前缀与实际的前缀相匹配,然后选择输入提示列表中的第一项。

表 9.1 QTextCursor API #1

方 法	说 明
<code>anchor()</code>	返回锚的位置;参考 <code>position()</code>
<code>atBlockEnd()</code>	如果光标在块的尾端,返回 <code>true</code>
<code>atBlockStart()</code>	如果光标在块的起始处,返回 <code>true</code>
<code>atEnd()</code>	如果光标在文档的尾端,返回 <code>true</code>
<code>atStart()</code>	如果光标在文档的起始处,返回 <code>true</code>
<code>beginEditBlock()</code>	告诉光标应该从撤销/重做的观点出发,把需要跟随的编辑动作当做一个单独的动作对待,参考 <code>charFormat()</code>
<code>block()</code>	返回包含光标的 <code>QTextBlock</code>
<code>blockCharFormat()</code>	为包含光标的块返回 <code>QTextCharFormat</code> ;参考 <code>charFormat()</code>
<code>blockFormat()</code>	为包含光标的块返回 <code>QTextBlockFormat</code>
<code>blockNumber()</code>	返回不包含表格或框架(不排除根框架)的文档中的光标所在行数,如 <code>QPlainTextEdit</code> 的 <code>QTextDocument</code>
<code>charFormat()</code>	为光标所在位置前紧靠的字符返回 <code>QTextCharFormat</code>
<code>clearSelection()</code>	移动锚到光标所在位置这样就没有任何内容被选中;参考 <code>removeSelectedText()</code>
<code>columnNumber()</code>	返回行中光标的位置
<code>createList(QTextListFormat)</code>	使用给定的格式(或给定的 <code>QTextListFormat::Style</code>)插入并返回 <code>QTextList</code> ,并把当前段定为列的第一项;参考 <code>insertList()</code>
<code>currentFrame()</code>	以 <code>QTextFrame *</code> 返回当前框
<code>currentList()</code>	以 <code>QTextList *</code> 返回当前列,如果当前光标位置不在一个列表中,则返回 0
<code>currentTable()</code>	以 <code>QTextList *</code> 返回当前表格,如果当前光标位置不在一个表格中,则返回 0
<code>deleteChar()</code>	如果存在选中的文本,删除它,否则,删除光标位置处的字符
<code>deletePreviousChar()</code>	如果存在选中的文本,删除它,否则,删除光标前的字符
<code>document()</code>	以 <code>QTextDocument *</code> 返回光标的文档

表 9.2 QTextCursor API #2

方 法	说 明
endEditBlock()	通知光标编辑由 beginEditBlock() 启动的动作序列已结束
hasComplexSelection()	如果选择内容不是一个简单的文本跨度, 如一个表格中的两个或更多的单元格返回 true,
hasSelection()	只要选择了任何东西, 返回 true
insertBlock(QTextBlockFormat, QTextCharFormat)	在光标位置插入一个新的空块, 还有两个重载, 一个仅包括一个块格式, 另一个则没有参数
insertFragment(QTextDocumentFragment)	在光标位置插入给定的文档片段
insertFrame(QTextFrameFormat)	以给定格式在当前光标位置插入一个 QTextFrame, 并将位置(和任何选择)移入框架
insertHtml(QString)	在当前位置插入 HTML 字符串
insertImage(...)	在光标位置插入一幅图像。有接受 QTextImageFormat、QString(filename) 和 QImage 的重载(可选择 QTextFrameFormat::Position)
insertList(QTextListFormat)	在光标位置插入一个新块, 使用给定格式(或给定的 QTextListFormat::Style)将它的第一项设为一个新的 QTextList, 返回列表; 参考 createList()
insertTable(int, int, QTextTableFormat)	用给定行数和列数和(可选择)格式插入并返回一个新的 QTextTable; 参考文本
insertText(QString, QTextCharFormat)	在光标位置使用(可选择)插入文本格式
isCopyOf(QTextCursor)	如果给定光标是当前光标的副本, 返回 true
isNull()	如果当前光标为空(也就是说, 没有用 QTextDocument 建立), 返回 true
joinPreviousEditBlock()	有效地“删除”最后的 beginEditBlock(), 以扩展前一个 beginEditBlock() 的范围
mergeBlockCharFormat(QTextCharFormat)	把当前块(或选定块)的字符格式与给定格式合并
mergeBlockFormat(QTextBlockFormat)	把当前块(或选定块)的字符格式与给定格式合并

表 9.3 QTextCursor API #3

方 法	说 明
mergeCharFormat(QTextCharFormat)	把当前光标位置字符(或选中)的格式与给定格式合并
movePosition(MoveOperation, MoveMode, int)	使用给定的操作通过(可选)计数次数移动光标位置。如果(可选)模式为 KeepAnchor, 锚不动, 创建一个选择内容; 默认模式为 MoveAnchor(移动选项在表 9.4 中列出)
position()	返回光标的位置; 参考 setPosition()
removeSelectedText()	删除选择的任何内容
select(SelectionType)	根据类型(Document、BlockUnderCursor、LineUnderCursor、WordUnderCursor 选择文本)
selectedTableCells(int *, int *, int *, int *)	用定义已选择的表格单元格的行数和列数, 第一列和列数填充指向自身的那些 int
selectedText()	以纯文本返回选择的文本
selection()	以 QTextDocumentFragment 返回选择的内容
selectionEnd()	返回以选择内容的结束位置
selectionStart()	返回以选择内容的起始位置
setBlockCharFormat(QTextCharFormat)	为当前块(或当前选择内容)设定字符格式; 参考 mergeBlockCharFormat()
setBlockFormat(QTextBlockFormat)	为当前块(或当前选择内容)设定块格式; 参考 mergeBlockFormat()
setCharFormat(QTextCharFormat)	为当前字符设定格式; 参考 mergeCharFormat()
setPosition(int, MoveMode)	移动光标到给定位置; 如果移动模式为 KeepAnchor, 光标不动, 创建一个选择内容; 参考 movePosition()
setVisualNavigation(bool)	如果设定为 true, 移动时跳过隐藏的段落; 默认为 false
visualNavigation()	返回设定为 bool 的视觉导航

如果仅有一个单独的输入提示,立即插入它,把 `true` 作为第二个参数传入 `insertCompletion()` 来提示当前的情况。否则调用 `QCompleter::complete()`,弹出输入提示列表——列表将具有一定宽度,还将放置在 `QRect` 定义的位置。`QPlainTextEdit::cursorRect()` 方法返回文本光标的矩形框,但这很明显太窄(几个像素),不能用做弹出列表的宽度。因此我们设定矩形框的宽度,让它足够显示弹出列表的第一(正常情况下只有一个)列,同时考虑一个垂直滚动条的宽度。

感谢我们前面建立了一个信号-槽连接,如果用户从弹出列表中选择一个输入提示,将(用默认为 `false` 的第二个参数,它意味着包含有多个单词,而不是一个单独的词的列表中的一个输入提示)调用 `insertCompletion()` 槽。

```
void XmlEdit::populateModel(const QString &completionPrefix)
{
    QStringList strings = toPlainText().split(QRegExp("\\W+"));
    strings.removeAll(completionPrefix);
    strings.removeDuplicates();
    qSort(strings.begin(), strings.end(), caseInsensitiveLessThan);
    model->setStringList(strings);
}
```

每次启动输入提示时,调用此方法。它动态地利用当前文档中的单词填充输入提示模型。这是通过提取文档的所有文本,并将其划分为单词列表完成的。然后,移除输入提示器前缀(如果存在的话)以及所有重复单词,再将单词排序,用新的列表取代模型现有的字符串。

为小的文档创建一个诸如此类完成提示单词列表是非常好的,但对于大的文档来说,这或许将带来很大的计算开销。还有,对一个空文档不建立任何的输入提示。一个可选择方法是在程序启动时,从字典中读取单词列表,并立即填充模型。另一个方法是使用一个如刚才描述的初始字典,增加所有新单词到这个字典中,这些新单词是用户在文档中输入的内容。在这两种方法中,模型都只建立一次且从不重新填充(但在第二个方法中将其更新)。

排序是关键——之前对 `QCompleter::setModelSorting(QCompleter::CaseInsensitivelySortedModel)` 的调用已告诉输入提示器模型是如何排序的(如果已排序)。因此,必须确保模型是按照我们要求的方式排序的。

一个类似于 `QList<T>`、`QStringList` 或 `QVector<T>` 的序列可以调用全局 `qSort()` 函数,如果想要对序列的一部分排序,或者(如此处一样)还想提供一个比较函数或仿函数,就需用开始和结束迭代器调用。`qSort()` 函数使用的序列必须提供 `operator<()` 方法,但 `QStrings` 默认的 `<` 执行大小写敏感的比较,因此我们必须自己提供微小的比较函数。

Qt 还提供 `qStableSort()` 函数,此函数的 API 与 `qSort()` 函数的相同。稳定排序和标准排序拥有同样的复杂程度,但在对两项或更多项进行比较时,保留相关项目顺序的复杂程度是一样的。

```
bool caseInsensitiveLessThan(const QString &a, const QString &b)
{
    return a.compare(b, Qt::CaseInsensitive) < 0;
}
```

由于 `QString::compare()` 方法不是语言环境感知的,如果像此处一样传递 `Qt::CaseInsensitive`,则该比较方法对大小写不敏感。

我们不愿意使用 `QString::localeAwareCompare()` 方法,因为它们是专门为用户可见字符串的排序列表设计的。但在撰写这些内容的时候,关于 `QCompleter` 对排序的理解仍然没有相关的文献可参考,我们使用了简单且更快速的 `QString::compare()` 方法。

```
void XmlEdit::insertCompletion(const QString &completion,
                               bool singleWord)
{

```

```

QTextCursor cursor = textCursor();
int numberOfCharsToComplete = completion.length() -
    completer->completionPrefix().length();
int insertionPosition = cursor.position();
cursor.insertText(completion.right(numberOfCharsToComplete));
if (singleWord) {
    cursor.setPosition(insertionPosition);
    cursor.movePosition(QTextCursor::EndOfWord,
        QTextCursor::KeepAnchor);
    completedAndSelected = true;
}
setTextCursor(cursor);
}

```

这个方法插入了需要运行输入提示的字符。它首先获取文档中的光标,然后计算需要插入的字符数——输入提示列表显示完整的单词,但我们仅想插入那些未输入的字符。再记录光标的位置(也就是输入提示字符将要插入的位置),插入完成单词所需要的那部分字符。

如果 `singleWord` 为 `true`,这意味着需要为单个单词提供输入提示,必须选择已插入字符,这样用户才能看到它们,并在想去掉它们时能很容易地去掉。在此情况下,我们把光标放回到插入动作发生的位置,然后使用 `QTextCursor::movePosition()` 方法,把光标移动到已完成单词的后面。这样就选择了已插入的字符(即将要讨论)。我们还设定 `completedAndSelected` 为 `true`,因为需要特别地应对下一个键的按下(或鼠标的点击),这样,用户就可以容易地接受或拒绝单个词的输入提示。

最后,设定文档的光标是已修改的光标,以让我们做出的改变(插入,也可能是选择)生效。

在文档中, `QTextCursor::movePosition()` 方法模拟用户导航,并拥有用于所有标准移动的枚举变量——这些移动动作如表 9.4 所列。

表 9.4 `QTextCursor::MoveOperation` 枚举变量

枚举变量	说 明
<code>Down</code>	移动光标到下一行
<code>End</code>	移动光标到文档的结尾处
<code>EndOfBlock</code>	移动光标到当前块的结尾处
<code>EndOfLine</code>	移动光标到当前行的结尾处
<code>EndOfWord</code>	移动光标到当前单词的结尾处
<code>Left</code>	把光标左移一个字符
<code>NextBlock</code>	移动光标到下一个块的开始处
<code>NextCell</code>	移动光标到表格的下一个单元格
<code>NextCharacter</code>	移动光标到下一个字符
<code>NextRow</code>	移动光标到表格下一行的第一个单元格
<code>NextWord</code>	移动光标到下一个单词的开始处
<code>NoMove</code>	不移动光标的位置
<code>PreviousBlock</code>	移动光标到上一个块的开始处
<code>PreviousCell</code>	移动光标到表格的上一个单元格
<code>PreviousCharacter</code>	移动光标到上一个字符
<code>PreviousRow</code>	移动光标到表格上一行的最后一个单元格
<code>PreviousWord</code>	移动光标到上一个单词的开始处
<code>Right</code>	把光标左移一个字符
<code>Start</code>	移动光标到文档的开始处
<code>StartOfBlock</code>	移动光标到当前块的开始处
<code>StartOfLine</code>	移动光标到当前行的开始处
<code>StartOfWord</code>	移动光标到当前单词的开始处
<code>Up</code>	移动光标到上一行
<code>WordLeft</code>	把光标向左移动一个单词的长度
<code>WordRight</code>	把光标向右移动一个单词的长度

`QTextCursor` 拥有两个位置, `QTextCursor::position()` (当前光标位置) 和 `QTextCursor::anchor()` (其他一些光标位置)。通常来说,位置与锚相同,但如果不同的话,它们两者中间的所有内容都被

选择。也就是说, `QTextCursor` 按照锚和位置定义了一个选择, 如果锚和位置相同的话, 则没有选择任何内容。

可以直接使用 `QTextCursor::setPosition()` 设定位置, 但是只能间接地设定锚。 `QTextCursor::movePosition()` 方法有两种运行模式(作为第二个参数传递), `QTextCursor::MoveAnchor`(默认)和 `QTextCursor::KeepAnchor`。如果选择 `KeepAnchor` 模式, 锚位置为移动前的所处位置或移动后所处位置。

注意, 我们的实施过程有个弱点(在一些使用情况下), 其原因在于, 它没有把输入文本匹配到输入提示文本。例如, 用户键入了“ali”, 选择输入提示单词为“AlignLeft”, 结果将是“alignLeft”。我们把修改此方法, 以匹配输入提示的情况留做练习。

```
void XmlEdit::keyPressEvent(QKeyEvent *event)
{
    if (completedAndSelected && handledCompletedAndSelected(event))
        return;
    completedAndSelected = false;

    if (completer->popup()->isVisible()) {
        switch (event->key()) {
            case Qt::Key_Up:      // Fallthrough
            case Qt::Key_Down:    // Fallthrough
            case Qt::Key_Enter:   // Fallthrough
            case Qt::Key_Return:  // Fallthrough
            case Qt::Key_Escape: event->ignore(); return;
            default: completer->popup()->hide(); break;
        }
    }
    QPlainTextEdit::keyPressEvent(event);
}
```

任何时候, 只要用户在 `XmlEdit` 组件中按下一个键, 就立刻调用这个事件处理器。我们已经重新实现了它, 以便处理两种特殊情况。第一, 如果用户已经完成了一个单个词的输入提示, 他们接下来按下的键(或是鼠标点击)将决定是接受输入提示, 还是拒绝。第二, 如果弹出菜单有效, 我们想忽略输入提示弹出菜单处理的任何按键。

如果 `completedAndSelected` 为 `true`, 我们就知道单个词的输入提示已经出现, 因此接下来用户将按下此键。在此情况下, 调用 `handledCompletedAndSelected()` 方法, 它返回一个布尔值, 以暗示该方法是否处理了按键动作, 如果已处理, 则返回; 如果没有处理, 将继续事件处理器。

如果输入提示器的弹出菜单是可见的, 忽略输入提示器处理(也就是说, 用上下箭头键来选择一个输入提示, 用 `Esc` 键取消和用 `Enter` 键接受)的按键动作。对于其他任何按下的键, 隐藏输入提示器, 这样就能取消输入提示, 把事件传递给基类的 `keyPressEvent()` 处理器, 并以正常方式对其进行处理。

还要注意的, 在所有情况下都把 `completedAndSelected` 的设为 `false`——只有在紧接着单个词之后的输入提示出现时, 它才为 `true`, 还有, 必须在下一次按下键或鼠标点击动作时将其设定为 `false`。

```
bool XmlEdit::handledCompletedAndSelected(QKeyEvent *event)
{
    completedAndSelected = false;
    QTextCursor cursor = textCursor();
    switch (event->key()) {
        case Qt::Key_Enter: // Fallthrough
        case Qt::Key_Return: cursor.clearSelection(); break;
        case Qt::Key_Escape: cursor.removeSelectedText(); break;
        default: return false;
    }
    setTextCursor(cursor);
    event->accept();
    return true;
}
```

`QTextCursor` 类用来实现通过编程创建并编辑 `QTextDocument`。通过调用 `QPlainTextEdit::textCursor()` 可以获取一个合适的光标;这与调用 `QTextCursor(document())` 具有同等效力,因为 `QPlainTextEdit::document()` 方法返回一个指针,此指针指向拥有其文本和格式的 `QTextDocument` (对于 `QTextEdit` 来说,与此处包括的其他所有事情一样,这些都是可实现的)。

`QTextCursor` 允许我们模拟用户的动作(在文档、插入的文本、删除的文本和选择的文本等之间进行导航)这通过它提供的 API 完成。在下一章中,将使用它实现编程创建 `QTextDocument` 时将对这个类的内容进行更多的介绍。为 `QTextCursor` 使用 `QPlainTextEdit` 和 `QTextEdit` 的方式非常简单:在文档中获取一个光标,使用 `QTextCursor` API 运行编辑动作,然后把修改的光标设定为文档的光标,以保证修改生效。

当用户完成单个字词后就立即按下某个按键,以调用 `handledCompletedAndSelected()` 方法。如果用户按下的是 `Enter` (或 `Return`) 键来接受输入提示,那么就清除该选择(因为对于单个字词来说,按下这些键就选择了输入提示,这与前面看到的结果相同)。如果用户按下 `Esc` 键来拒绝输入提示,那么就删除所选文字。在以上这两种情况下,我们对事件调用 `accept()` 来告诉 Qt,我们已经做出了处理,并返回 `true` 以使 `keyPressEvent()` 方法重新生效,这就意味着不需再做其他处理了。

如果用户键入了其他内容,则返回 `false`,用户键入的内容就会得到处理。例如,用户键入一个字母,基类的按键处理器将删除选择的文本,并插入键入的字母。在大部分编辑窗口部件(不仅是 Qt 提供的那些)中已选择的文本上输入内容时都将执行这样的动作。这意味着用户继续输入内容就能拒绝单个词的输入提示。

```
void XmlEdit::mousePressEvent(QMouseEvent *event)
{
    if (completedAndSelected) {
        completedAndSelected = false;
        QTextCursor cursor = textCursor();
        cursor.removeSelectedText();
        setTextCursor(cursor);
    }
    QPlainTextEdit::mousePressEvent(event);
}
```

如果单个词的输入提示刚刚出现, `completedAndSelected` 为 `true`,像我们刚才看到的那样,用户可以使用键盘接受或拒绝输入提示。考虑到输入提示弹出菜单的一致性,用户只要用鼠标点击文档的其他地方就可以拒绝输入提示。移除输入提示文本的代码与在我们刚才看到的 `handledCompletedAndSelected()` 方法中使用的代码相同。

无论单个词的输入提示被接受还是拒绝,一旦用户按下了一个键或是点击了鼠标, `completedAndSelected` 就一定被设定为 `false`,这样就能正确地处理随后的按键动作了。

现在,我们已经介绍完了输入提示。如前面所看到的那样,为 `QLineEdit` 和 `QComboBox` 建立输入提示可以很容易地通过调用它们的 `setCompleter()` 方法实现。但对于多行窗口部件,如 `QPlainTextEdit` 或 `QTextEdit`,还有更多的工作需要完成。相比之下,建立当前行和语法的加亮非常简单,虽然后者的加亮需要用到正则表达式,下一小节我们将看到这些内容。

9.2.2.2 语法加亮

在此小节中,我们主要把重点放在语法加亮上,首先要看一下如何加亮当前行。在前一小节中,已看到一个来自 `QPlainTextEdit` 的 `cursorPositionChanged()` 信号,它是连接到了自定义 `highlightCurrentLine()` 槽。这个连接保证了无论何时光标被移除(无论是通过键盘还是鼠标),都调用这个槽。

```
void XmlEdit::highlightCurrentLine()
{
    QList<QTextEdit::ExtraSelection> extraSelections;
    QTextEdit::ExtraSelection selection;
    QBrush highlightColor = palette().alternateBase();
    selection.format.setBackground(highlightColor);
    selection.format.setProperty(QTextFormat::FullWidthSelection,
                                true);
    selection.cursor = textCursor();
    selection.cursor.clearSelection();
    extraSelections.append(selection);
    setExtraSelections(extraSelections);
}
```

从 Qt 4.2 开始, QPlainTextEdit 和 QTextEdit 类已经支持通过编程添加额外选择的功能了。它的主要用途是提供额外的加亮,如当前行加亮或有断点的行的加亮等。

这个方法首先创建了一个选择列表(对刚才添加的项),然后设定选择内容的 QTextCharFormat 的背景颜色,这也是它的属性之一。QTextFormat 类(QTextCharFormat 的基类)提供了一个特性机制,它能够在不损害二进制兼容性的前提下让在未来的 Qt 版本中添加更多特性变得很容易。它的一些属性是为段落(也就是为 QTextBlockFormat)而设的,另外一些是为字符而设,还有一些是同时为这两者而设的。此处我们已说过,选择内容的格式需要应用到它的完整宽度。

建立选择内容的格式,同时提取文档的光标,并清除现有的选择(这就是把锚移动到了当前光标位置。然后设定额外的选择)包括刚才已创建的选择。此时我们还不能像平常期待的那样看到任何选择内容——锚和位置都在相同的地方,因此它们中间没有可选择的东西。尽管如此,开启 QTextFormat::FullWidthSelection 属性就可以保证选择内容有效地被扩展到了光标所在行的全部宽度,这和锚没有任何关系。

我们需要的信号-槽链接以及 highlightCurrentLine() 方法都是用来加亮当前行的。相比之下,为了提供语法加亮,我们只需要用一个指向 QTextDocument 的指针创建 QSyntaxHighlighter 子类的实例就行了——当然,还必须创建子类(我们已在前面看到过创建的实例)。

此处的 Qt 示例和源代码包括一些现成的 QSyntaxHighlighter 子类。在 examples/richtext/syntaxhighlighter 示例中,有一个用于 C++/Qt 代码的语法加亮器。所有 examples/xmlpatterns 中的示例都有一个 XML 语法加亮器,它比我们这里将要介绍的这个要短一些、简单一些。另外,Qt Designer 的源代码 tools/designer/src/lib/shared 包括三个语法加亮器,CSS(层叠样式表,Cascading Style Sheets)使用的 csshighlighter.cpp、HTML 使用的 htmlhighlighter.cpp 和 JavaScript 使用的 qscripthighlighter.cpp。这些都不是 Qt 的公共 API,但它们可以作为良好的开端。其他可以考虑的是 GNU Source-Highlight library(www.gnu.org/software/src-highlight)。这个库为大范围的语言、格式提供了语法加亮,Source-Highlight Qt library(srchiliteqt.sourceforge.net)为它提供了 QSyntaxHighlighter 外壳。

语法加亮最便捷的方法是使用正则表达式(regexes),它也是此处采用的方法之一,假定读者已具有正则表达式的基本知识^①。在大文档中使用正则表达式将会导致不良的运行效果(这或许也是前面提到的 htmlhighlighter.cpp 不使用它们的原因)。

QSyntaxHighlighter 子类必须提供 highlightBlock() 方法的重实现。调用这个方法能为单行文

^① 在 qt.nokia.com/doc/qregexp.html 中的 QRegExp 文献提供了正则表达式的简要介绍。一本关于正则表达式的好书是 Jeffrey E. F. Friedl 的《掌握正则表达式》(Mastering Regular Expressions), ISBN 0596528124——这本书中没有明确地讲述到 QRegExp 的内容,但却包含了 Perl 正则表达式,它们两者都是相似的(虽然后者有更多的功能)。

本提供加亮,在需要的时候,它会自动调用。对于那些跨度为多行的语法(如多行注释),通常有必要维护一些状态,这样才能决定一个给定行是否在多行中。为了支持这些功能,我们可以用 `setCurrentBlockState()` 方法连接一个整数(如一个状态的 ID)到每一行,这样就可以用 `previousBlockState()` 方法找出前一行的状态了。

对于 `XmlEdit` 的 `XmlHighlighter` 类,支持两种状态、四种语法元素类型和使用五种不同字符格式的加亮。我们使用的状态、类型和格式都是 `xmlhighlighter.hpp` 中的私有数据:

```
enum State {Normal=0x01, InComment=0x02};
enum Type {Tag, Attribute, Entity, Comment};

QTextCharFormat tagFormat;
QTextCharFormat attributeNameFormat;
QTextCharFormat attributeValueFormat;
QTextCharFormat entityFormat;
QTextCharFormat commentFormat;
QMultiHash<Type, QRegExp> regexForType;
```

我们已使用 `QMultiHash` 把一个或更多正则表达式和每一个元素类型连接了起来。无论一个正则表达式在什么地方与元素匹配,都为它的类型应用合适的格式(前一章介绍过 `QMultiHash`)。

现在,将介绍构造方法、`highlightBlock()` 方法,还要介绍它们支持的方法,以说明语法加亮是如何实现的。

```
XmlHighlighter::XmlHighlighter(QTextDocument *parent)
    : QSyntaxHighlighter(parent)
{
    tagFormat.setForeground(Qt::darkBlue);
    attributeNameFormat.setForeground(Qt::blue);
    attributeValueFormat.setForeground(Qt::darkYellow);
    entityFormat.setForeground(Qt::darkRed);
    commentFormat.setForeground(Qt::darkGreen);
    commentFormat.setFontItalic(true);

    addRegex(Tag, "<[!]?\\w+(?:/>)?", false);
    addRegex(Tag, "(?:</\\w+)?[?]>");
    addRegex(Attribute, "(\\w+(?:\\w+)?)=(\\\"[^\"]+\\\"|'[^']*'+)");
    addRegex(Entity, "&({?:?#\\d+|\\w+};)");
    addRegex(Comment, "<!--.*-->");
}
```

在此处的构造函数中,我们把将应用加亮的 `QTextDocument` 传递给基类,还建立了 `QTextCharFormat`——在大部分情况下仅设定一个前景文字颜色,但对于注释,还把它设为斜体。当然,我们可以应用其他任何格式,只要 `QTextCharFormat` 支持。

对于每一个 XML 元素,我们想要让其语法加亮,使用自定义 `addRegex()` 帮助方法可以添加一个正则表达式到 `QMultiHash`。帮助的第三个参数默认为 `true`(它意味着使用最小的,也就是非贪婪匹配)。

第一个正则表达式用来匹配标记的开始部分或自身不完整的的标记。例如, `< tag`、`< ! tag`、`< ? tag` 或 `< tag /`。第二个正则表达式用来匹配关闭标记或一个标记的关闭部分,如 `< /tag >`、`? >` 或者 `>`。第三个正则表达式匹配 `key = value` 属性,如 `key = "value"`。严格地讲,属性如果在一个标记内,它才加亮,但此处我们采用一个更单纯的方法,只要遇到的 `key = value`,就加亮它。第四个正则表达式用来匹配实体,如 `´`; 或者 `é`;。最后一个正则表达式用来匹配一个单独行内的注释——我们将简短介绍如何匹配多行跨度的注释。除了第一个以外,所有的正则表达式都使用最小匹配。

```
void XmlHighlighter::addRegex(Type type, const QString &pattern,
                               bool minimal)
{
    QRegExp regex(pattern);
    regex.setPatternSyntax(QRegExp::RegExp2);
    regex.setMinimal(minimal);
    regexForType.insert(type, regex);
}
```

这个帮助方法创建了一个 QRegExp,并用类型作为它的键,把它插入到 QMultiHash。

无论什么时候使用捕捉括号,总是调用 QRegExp::setPatternSyntax(QRegExp::RegExp2)。这保证了,当使用非最小(也就是贪婪的)匹配时,调用行为与其他一些正则表达式引擎(特别是 Perl 的)类似,而不是 QRegExp 自己相当奇特的行为(更多类 Perl 行为将如期出现在 Qt5 中)。

```
void XmlHighlighter::highlightBlock(const QString &text)
{
    setCurrentBlockState(Normal);
    highlightPatterns(text);
    highlightComments(text);
}
```

默认情况下,当前行的状态为 Normal(虽然可能会改变它)。我们已分解出独立的两种方法来实现语法加亮。使用正则表达式的 highlightPatterns() 方法和 highlightComments() 方法,前者在一个单独行的上下文中工作,后者使用字符串搜索,并处理多行跨度的注释。我们必须第二次调用 highlightComments(),因为想要加亮注释掉的 XML,以覆盖其他加亮。

```
void XmlHighlighter::highlightPatterns(const QString &text)
{
    QHashIterator<Type, QRegExp> i(regexForType);
    while (i.hasNext()) {
        i.next();
        Type type = i.key();
        const QRegExp &regex = i.value();
        int index = regex.indexIn(text);
        while (index > -1) {
            int length = regex.matchedLength();
            if (type == Tag)
                setFormat(index, length, tagFormat);
            else if (type == Attribute) {
                setFormat(index, regex.pos(2) - index - 1,
                           attributeNameFormat);
                setFormat(regex.pos(2) + 1, regex.cap(2).length() - 2,
                           attributeValueFormat);
            }
            else if (type == Entity)
                setFormat(index, length, entityFormat);
            else if (type == Comment)
                setFormat(index, length, commentFormat);
            index = regex.indexIn(text, index + length);
        }
    }
}
```

这个方法的原理很简单:遍历 QMultiHash 中的每一个正则表达式,应用正则表达式到当前行(文本中),如果匹配,应用合适的加亮到正则表达式相关的 Type。

一些键拥有多个值,这没有关系;迭代器将返回键,返回的键与键-值对(key-value pair)的个数相对应,这些值保存在哈希表中。只要有一个正则表达式匹配时,就为大部分元素应用相应的 QTextCharFormat,将其作为匹配的长度。对于属性则是个例外,我们分别单独地格式化键和值的部分。注意,虽然第一个对 QRegExp::indexIn() 的调用从文本的起始处开始(因为没有给定补偿),接下来在 while 循环中间的调用将在上一个匹配结尾后得到补偿。

以下是正则表达式中的一个匹配注释,此注释在同一行开始,并结束。highlightComments() 方法用来为多行跨度的注释提供加亮。

```
void XmlHighlighter::highlightComments(const QString &text)
{
    const QString StartOfComment("<!--");
    const QString EndOfComment("-->");

    if (previousBlockState() > -1 &&
        (previousBlockState() & InComment) == InComment) {
        int end = text.indexOf(EndOfComment);
        if (end == -1) {
            setFormat(0, text.length(), commentFormat);
            setCurrentBlockState(currentBlockState() | InComment);
            return;
        }
        else
            setFormat(0, end + EndOfComment.length(), commentFormat);
    }

    int start = text.lastIndexOf(StartOfComment);
    if (start != -1) {
        int end = text.lastIndexOf(EndOfComment);
        if (end < start) {
            setFormat(start, text.length(), commentFormat);
            setCurrentBlockState(currentBlockState() | InComment);
        }
    }
}
```

默认块的 state 值为 -1,通常为 4 个自定义块状态使用正整数,在本示例中,分别为 Normal 和 InComment 状态使用 0x01 和 0x02。我们选择使用十六进制数定义值,这样可以让它变得更加明显,也就能在它们上使用按位运算符了。但这并不是本示例真正想要的效果,更复杂的加亮器来说或许会有更多状态(0x04、0x08、0x10,等等),并且或许要能将状态合并(也就是说,0x01 | 0x04 以产生一个包含状态 1 和 4 的 0x05 值)。

此方法必须考虑的情况包括:

1. 已经处于一个多行注释中,且注释在或者不在当前行结束。
2. 一个多行注释在此行开始。
3. 以上情况以外的情况。

如果我们已经处于一个多行注释中, QSyntaxHighlighter::previousBlockState() 将包含 InComment 状态。因为我们想要支持状态合并,而不是直接用 InComment 比较前一个块的状态,所以用 InComment 屏蔽上一个块状态,然后再进行比较。此方法允许我们随意删除状态,即便两个或更多的状态已结合。如果已处于一个多行注释中,检查注释是否在此行结束。如果不在此行结束,应用注释格式到此整行,并把 InComment 状态和当前块的状态合并(前面为 Normal 设定的),然后返回,因为没有更多的任务需要完成。但如果注释在此行结束,为注释从头到尾应用注释格式,然后继续此方法,因为有可能有一个新的多行注释继续在此行开始。

如果我们不在一个多行注释中或注释已在当前行结束,检查此行是否有新的注释开始。如果发现新的注释,查找它的结尾。必须要忽略在本行开始并结束的注释,因为,我们已经在其他地方对它们进行了格式化,如果在一个注释开始元素之前发现结束元素(也就是说,找到的与前一个注释关联的结束元素,不是在此行开始的多行注释的结尾),就要小心谨慎了。如果一个多行注释没有在此行开始,从注释开始,到此行结束都应用注释格式,并合并 InComment 状态与当前状态。

在其他任何情况下不做任何处理,因为,要么此处没有格式需要处理,要么已经用 highlightPatterns() 方法处理过格式了。

到目前为止,我们就已经完成了对 XmlHighlighter 类和语法加亮的介绍。我们为加亮使用了正则表达式,还用简单的字符串来查找一些正则表达式。另一个方法是实现一个语法分析器,并用它来逐个字符地读取文本。如果状态整数或比特模式不充分,可以通过连接自定义数据以及每一行来对它进行补充(使用 `QSyntaxHighlighter::setCurrentBlockUserData()`)。

当然,有些时候它不是我们想要的高亮效果,就要为用户提供用户进入所选择文本的颜色、字体和字体属性的功能。在下一节中,我们将介绍如何为单行编辑器,以及多行编辑器实现这样的效果。

9.3 一个单行的富文本编辑器

Qt 已经提供了一个窗口部件,此部件可以用来编辑富文本(`QTextEdit`),但它的设计目标是编辑多行文本,而不是我们平时想要的单行文本。尽管如此,在只允许显示并编辑单个行时,我们可以通过子类化 `QTextEdi` 利用所有它提供的编辑以及渲染功能的优点。当然,我们还可以使用 `QTextEdit` 子类化 `QPlainTextEdit`,它更为便捷,因为它提供了很多有用的槽(如 `QTextEdit::setFontItalic()`),而 `QPlainTextEdit` 却没有提供这些槽。

在本章中,我们将创建一个 `RichTextLineEdit` 窗口组件,它可以用来编辑文本的单个行,并允许用户应用文本效果到单个字符或文本中的单词,如粗体、斜体或设定颜色。`RichTextLineEdit` 由 `Timelog` 应用程序的 `RichTextDelegate` 使用,如在第 5 章看到的那样。我们将为 `RichTextLineEdit` 提供工具提示和上下文菜单,如图 9.4 所示。

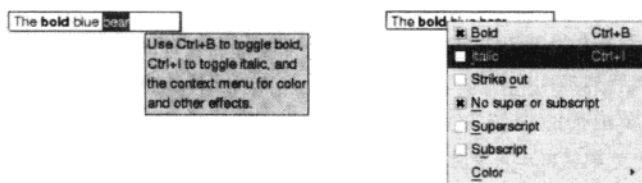


图 9.4 `RichTextLineEdit` 的工具提示和上下文菜单

首先,来看一段从类的定义中提取的代码,但此处没有给出大部分的私有槽、方法或是数据(虽然将在介绍公共和受保护的方法和槽的时候介绍这些所有内容)。

```
class RichTextLineEdit : public QTextEdit
{
    Q_OBJECT

public:
    explicit RichTextLineEdit(QWidget *parent=0);
    QString toSimpleHtml() const;
    QSize sizeHint() const;
    QSize minimumSizeHint() const;

public slots:
    void toggleItalic() { setFontItalic(!fontItalic()); }
    void toggleBold() { setFontWeight(fontWeight() > QFont::Normal
        ? QFont::Normal : QFont::Bold); }

signals:
    void returnPressed();

protected:
    void keyPressEvent(QKeyEvent *event);
    ...

private:
    enum Style {Bold, Italic, StrikeOut, NoSuperOrSubscript,
        Subscript, Superscript};
    ...
};
```

`setFontItalic()` 和 `setFontWeight()` 方法是从 `QTextEdit` 继承的,但 Qt 可以处理一些不同的字体粗细,我们已把粗体选项缩减为一个简单的 on/off。

`returnPressed()` 信号的设计目的是把 `RichTextLineEdit` 变得更像 `QLineEdit`,帮助委托(或任何连接到委托的对象)了解到,用户已结束并确认他们做出的修改。`keyPressEvent()` 的重新实现是探测 Enter 和 Return 并发射信号的地方。

既然已经子类化了 `QTextEdit`,我们就不必自己去为任何东西着色,标准行为的大部分(如复制和粘贴,还有撤销/重做)都会得到继承。这意味着,唯一要做的就是确保 `RichTextLineEdit` 被设定为只显示一行,并且能提供我们想要的其他自定义行为。和往常一样,我们从构造函数开始介绍。

```
RichTextLineEdit::RichTextLineEdit(QWidget *parent)
    : QTextEdit(parent)
{
    setLineWrapMode(QTextEdit::NoWrap);
    setWordWrapMode(QTextOption::NoWrap);
    setAcceptRichText(true);
    setTabChangesFocus(true);
    setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    createShortcuts();
    createActions();
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, SIGNAL(customContextMenuRequested(const QPoint&)),
            this, SLOT(customContextMenuRequested(const QPoint&)));
}
```

使用构造函数做出所有我们想要的明显改变:关闭行和词包装,HTML 将作为富文本接受,这样做可以确认它是不是从剪贴板粘贴过来的,让 Tab 键移动焦点,而不是插入一个制表符,关闭滚动条。

提供一个上下文菜单最简单的方式是添加 `QAction` 到一个窗口部件,并以此设定 `Qt::Actions-ContextMenu` 的上下文菜单策略,把这些任务留给 Qt 来完成。对于此特别的窗口组件来说,这或许不是适合的解决办法,因为上下文菜单启动后,我们想在任意时间更新动作的选择状态。我们选择设定 `RichTextLineEdit` 的方法是设定 `Qt::CustomContextMenu` 的一个上下文菜单策略,并以此连接它发射的 `customContextMenuRequested()` 信号,此信号是在启动了一个上下文菜单后发射的,这个设定的策略会让一个自定义槽生效,这个自定义槽与我们自己建立并显示上下文菜单时使用的槽同名。

```
void RichTextLineEdit::createShortcuts()
{
    QShortcut *boldShortcut = new QShortcut(QKeySequence::Bold,
        this, SLOT(toggleBold()));
    QShortcut *italicShortcut = new QShortcut(QKeySequence::Italic,
        this, SLOT(toggleItalic()));

    setToolTip(tr("<p>Use %1 to toggle bold, %2 to toggle italic, "
        "and the context menu for color and other effects.")
        .arg(boldShortcut->key().toString(
            QKeySequence::NativeText))
        .arg(italicShortcut->key().toString(
            QKeySequence::NativeText)));
}
```

我们通过创建合适的 `QShortcut`,并使用标准的 `QKeySequence` 来为粗体和斜体提供键盘快捷键。在 Linux 和 Windows 下,粗体动作的快捷键是 Ctrl + B,但在 Mac OS X 下则为⌘ + B^①。

① 在 Windows 和 Linux 下,很多标准快捷键都使用 Ctrl 键(Ctrl + C 实现复制),而在 Mac OS X 下,则使用⌘键(例如,⌘ + C)。有鉴于此,当第一次把 Qt 导入到 Mac OS X 用做简单的跨平台开发,Qt 会简单“交换”(swap)Ctrl 键和⌘键——也就是说,为什么在 Mac OS X 下按下⌘ + X 与在其他操作系统下按下 Ctrl + X 的功能会相同。事后看来,这可能并不是一个明智的选择,另外,从 Qt 4.6 开始,可以通过调用 `QApplication::setAttribute(Qt::AA_MacDontSwapCtrlAndMeta)` 来避免键的交换。

一旦拥有了快捷键,创建一个工具提示,用它来显示这些快捷键——将它们转换成字符串的形式,这样就可以在应用程序所运行平台上正确地显示了。

```
void RichTextLineEdit::createActions()
{
    boldAction = createAction(tr("Bold"), Bold);
    ...
    colorAction = new QAction(tr("Color"), this);
    colorAction->setMenu(createColorMenu());

    addActions(QList<QAction*>() << boldAction << italicAction
        << strikeOutAction << noSubOrSuperScriptAction
        << superScriptAction << subScriptAction << colorAction);
    AQP::accelerateActions(actions());
}
```

这个方法用来创建动作——我们已忽略了大部分的动作,因为几乎所有的动作都是以相同的方式建立的。即将要介绍的 `createAction()` 方法有两个参数:菜单中要显示的文本和一个数据项——此处为一个枚举变量。

颜色动作是不同寻常的,因为它不需要连接到任何东西,而是获取一个菜单。

最后,添加所有动作到 `RichTextLineEdit`;我们可以仅把它们存储在一个私有 `QList<QAction*>` 中。然后给予它们键盘快捷键。

```
QAction *RichTextLineEdit::createAction(const QString &text,
                                         const QVariant &data)
{
    QAction *action = new QAction(text, this);
    action->setData(data);
    action->setCheckable(true);
    action->setChecked(false);
    connect(action, SIGNAL(triggered()), SLOT(applyTextEffect()));
    return action;
}
```

这是个简便的方法,它创建我们需要的动作,将所有动作设为可选项,并初始为未选择。这些所有的动作都连接到 `applyTextEffect()` 方法。

```
QMenu *RichTextLineEdit::createColorMenu()
{
    QMenu *colorMenu = new QMenu(this);
    QPixmap pixmap(22, 22);
    typedef QPair<QColor, QString> ColorPair;
    foreach (const ColorPair &pair, QList<ColorPair>())
        << qMakePair(QColor(Qt::black), tr("Black"))
        ...
        << qMakePair(QColor(Qt::darkRed), tr("Dark Red"))) {
        pixmap.fill(pair.first);
        QAction *action = colorMenu->addAction(pixmap, pair.second);
        action->setData(pair.first);
    }
    connect(colorMenu, SIGNAL(triggered(QAction*)),
        this, SLOT(applyColor(QAction*)));
    AQP::accelerateMenu(colorMenu);
    return colorMenu;
}
```

此方法用来创建颜色动作使用的颜色菜单,如图 9.5 所示。首先在堆栈上创建一个新的 `QMenu`,然后创建一对列表(一些 `QColor` 和它们的名称),然后立即遍历。对于每一个对,使用相应颜色的一个像素映射创建一个拥有颜色名称的动作,然后设定动作数据到 `QColor` 值(`Qt` 的 `QPixmap` 的工作原理与类相似,因为它们使用写时复制,这样每一个

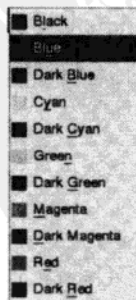


图 9.5 `RichTextLineEdit` 的颜色子菜单

动作都获得自己唯一的像素映射。第 12 章将介绍如何创建颜色卡,它比此处使用的纯方块更漂亮)。

连接菜单的 `triggered()` 动作到 `applyColor()` 槽——被传递的 `QAction *` 与用户选择的菜单项相对应。最后为菜单项创建键盘加速键,返回菜单到调用方。

此处需要使用 `typedef`,因为 Qt 的 `foreach` 宏有 `foreach (item, sequence)` 语法。逗号是语法的一部分,用来分辨项和序列(如果序列有逗号也是没有关系的;只有第一个逗号是重要的)。因此对于项对 (`item pair`),要么必须使用 `typedef`,要么就必须首先创建列表和列;例如:

```
QList<QPair<QColor, QString> > pairList;
pairList << qMakePair(QColor(Qt::black), tr("Black"))
...
    << qMakePair(QColor(Qt::darkRed), tr("Dark Red"));
QPair<QColor, QString> pair;
foreach (pair, pairList)
...

```

使用这种方法需要稍微多一点代码,但很明显,如果想独立地创建对(例如为了后面再次使用它们)而不是在使用的地方,使用这种方法更加方便。

```
void RichTextLineEdit::applyColor(QAction *action)
{
    Q_ASSERT(action);
    setTextColor(action->data().value<QColor>());
}

```

当用户从上下文菜单的颜色子菜单中选择一个颜色时,就调用这个槽。它仅调用基类的 `setTextColor()` 方法,用以设定选择的颜色。

`QVariant` 为 QtCore 库中的那一类提供了设置方法——例如, `QVariant::toDate()`、`QVariant::toPoint()`、`QVariant::toString()`,等等。但对于其他 Qt 库中的类,必须使用 `QVariant::value < T > ()` 设置方法, `T` 为我们想要提取的类型。

```
void RichTextLineEdit::applyTextEffect()
{
    if (QAction *action = qobject_cast<QAction*>(sender())) {
        Style style = static_cast<Style>(action->data().toInt());
        QTextCharFormat format = currentCharFormat();
        switch (style) {
            case Bold: toggleBold(); return;
            case Italic: toggleItalic(); return;
            case StrikeOut:
                format.setFontStrikeOut(!format.fontStrikeOut());
                break;
            case NoSuperOrSubscript:
                format.setVerticalAlignment(
                    QTextCharFormat::AlignNormal);
                break;
            case Superscript:
                format.setVerticalAlignment(
                    QTextCharFormat::AlignSuperScript);
                break;
            case Subscript:
                format.setVerticalAlignment(
                    QTextCharFormat::AlignSubScript);
                break;
        }
        mergeCurrentCharFormat(format);
    }
}

```

如果用户启动了上下文菜单(除了颜色菜单中的那些颜色)中的任何动作,那么调用这个槽。首先



把启动的对象强制转换为一个 QAction 指针(它应经常工作),然后提取动作数据。在这种情况下,为 RichTextLineEdit 类的私有 Style enum 的值。然后使用这个值决定需要应用的格式。对于粗体和斜体来说,只需要触发当前设定开关就可以返回。但对于设定加删除线或垂直对准,必须更新当前 QTextCharFormat 的副本,然后把此副本与当前格式合并,这样才能让更改生效。

QObject::sender() 方法返回启动槽(如果槽被作为一个方法调用,则为 0)QObject,在此情况下,只有 QAction 调用槽。我们只要使用 dynamic_cast<>(),而不是 qobject_cast<>(),但我们倾向于为那些 QObject 使用 qobject_cast<>(),因为它不需要 RTTI(运行时类型信息,Run Time Type Information)的支持就可以工作,并且可以跨越动态库边界工作。一个可选择方法是使用 sender(),以进一步使用 QSignalMapper;这样将提供更好的封装,且只要付出稍微增加代码的代价。当然,我们还可以为每一个动作创建单独的槽。

```
void RichTextLineEdit::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Enter ||
        event->key() == Qt::Key_Return) {
        emit returnPressed();
        event->accept();
    }
    else
        QTextEdit::keyPressEvent(event);
}
```

仅为了提供 returnPressed() 信号,重新实现这个处理器,否则,忽略 Enter 或 Return 按键动作。其他所有键的按下将被传送到基类。

```
void RichTextLineEdit::customContextMenuRequested(const QPoint &pos)
{
    updateContextMenuActions();

    QMenu menu(this);
    menu.addAction(actions());
    menu.exec(mapToGlobal(pos));
}
```

当用户启动上下文菜单(通过右键点击或通过使用一个平台定义的键序列)时将在构造函数中建立一个信号-槽连接,并调用这个槽。首先调用 updateContextMenuActions(),以适当地设定每一个动作的选择状态。然后创建一个菜单,并为它添加窗口部件的动作(前面创建的)。最后弹出菜单,使用 QMenu::exec() 把相对窗口部件的位置转换为相对屏幕(也就是全局)的位置。

如果用户选择颜色子菜单中的一个颜色,调用前面看到的 applyColor() 方法。同样,如果选择了其他项,调用 applyTextEffect() 方法。

```
void RichTextLineEdit::updateContextMenuActions()
{
    boldAction->setChecked(fontWeight() > QFont::Normal);
    italicAction->setChecked(fontItalic());
    const QTextCharFormat &format = currentCharFormat();
    strikeOutAction->setChecked(format.fontStrikeOut());
    noSubOrSuperScriptAction->setChecked(format.verticalAlignment() ==
        QTextCharFormat::AlignNormal);
    superScriptAction->setChecked(format.verticalAlignment() ==
        QTextCharFormat::AlignSuperScript);
    subScriptAction->setChecked(format.verticalAlignment() ==
        QTextCharFormat::AlignSubScript);
}
```

此方法用来更新上下文菜单的动作,以便让它们反应光标位置处的文本状态。对于所有较长的行,代码非常简单,根据当前光标位置处的文本状态来决定选择或者不选择某个动作。

```

QSize RichTextLineEdit::sizeHint() const
{
    QFontMetrics fm(font());
    return QSize(document()->idealWidth() + fm.width("W"),
        fm.height() + 5);
}

```

尺寸提示的理想值是窗口部件的未来尺寸。我们已创建了一个尺寸,实际上,它是根据纯文本内容建立的(因为未显示的 HTML 标签将篡改计算,就像在介绍 `RichTextDelegate::sizeHint()` 方法时看到的那样),且已添加一个“W”字符的宽度,用它来实现一个字节的水平边距。同样,垂直方向的尺寸为实际尺寸加 5 像素,这样就能生成一些垂直边距。

```

QSize RichTextLineEdit::minimumSizeHint() const
{
    QFontMetrics fm(font());
    return QSize(fm.width("WWWW"), fm.height() + 5);
}

```

最小的尺寸提示为实际最小尺寸,Qt 将总是缩放窗口部件到此最小尺寸。此处,设定最小尺寸提示为 4 个“W”字符的宽度,它与为 `sizeHint()` 使用的尺寸相同。

现在,我们已介绍了所有 `RichTextLineEdit` 真正需要的方法。并且我们还另外添加了自定义方法,我们将讨论此自定义方法。

`QTextEdit` 基类有一个 `toHtml()` 方法,此方法返回 HTML 格式中的文本。我们已选择忽略它并提供自己的 `toSimpleHtml()`,因为我们仅需要 `QTextEdit` 一个非常小的子集,它可以处理 HTML,通过这种方法的限制可以产生最紧凑的 HTML。基于这样的考虑,如果 HTML 文本为 `The bold blue bear`,`QTextEdit::toHtml()` 方法将返回接下来的 HTML(用一些换行符代替空白,这样它就能适合于显示在页面上):

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html40/strict.dtd">
<html><head><meta name="qrichtext" content="1" />
<style type="text/css">p, li { white-space: pre-wrap; }</style>
</head><body style=" font-family:'Nimbus Sans L';
font-size:12pt; font-weight:400; font-style:normal;">
<p style=" margin-top:0px; margin-bottom:0px; margin-left:0px;
margin-right:0px; -qt-block-indent:0; text-indent:0px;">The
<span style=" font-weight:600;">bold</span>
<span style=" color:#0000ff;">blue</span> bear</p>
</body></html>

```

实际上,输出内容有 545 个字符,虽然在实践中此数字将稍有变化,因为默认字体(此处为 `Nimbus Sans L`)将可能在其他机器上有所不同。另外,输出可能会因 Qt 版本的不同,而有所差异。将此与 `toSimpleHtml()` 产生的 HTML 相比较:

```
The <b>bold</b> <font color="#0000ff">blue</font> bear
```

此处仅有 54 个字符[严格地讲,这不是有效的 HTML 输出;如果想让它有效,至少需要一个 `DOCTYPE`(文档类型)声明,还有 `<html>`、`<head>` 和 `<body>` 标记]当然,`toSimpleHtml()` 方法具有很大的局限性,`toHtml()` 方法则具有更强大的功能,但对于简单 HTML 的单行来说,使用更加紧凑的格式显然是不可取的。

```

QString RichTextLineEdit::toSimpleHtml() const
{
    QString html;
    for (QTextBlock block = document()->begin(); block.isValid();
        block = block.next()) {
        for (QTextBlock::iterator i = block.begin(); !i.atEnd();
            ++i) {

```

```

QTextFragment fragment = i.fragment();
if (fragment.isValid()) {
    QTextCharFormat format = fragment.charFormat();
    QColor color = format.foreground().color();
    QString text = Qt::escape(fragment.text());
    QStringList tags;
    if (format.verticalAlignment() ==
        QTextCharFormat::AlignSubScript)
        tags << "sub";
    else if (format.verticalAlignment() ==
        QTextCharFormat::AlignSuperScript)
        tags << "sup";
    if (format.fontItalic())
        tags << "i";
    if (format.fontWeight() > QFont::Normal)
        tags << "b";
    if (format.fontStrikeOut())
        tags << "s";
    while (!tags.isEmpty())
        text = QString("<%1>%2</%1>")
            .arg(tags.takeFirst()).arg(text);
    if (color != QColor(Qt::black))
        text = QString("<font color=\"%1\">%2</font>")
            .arg(color.name()).arg(text);
    html += text;
}
}
return html;
}

```

我们需要遍历所有存储在基类的 `QTextDocument` 内部的所有文本,并在考虑每个文本格式属性的情况下,输出相应的 HTML。`QTextDocument` 类使用一个类树结构层级,此层级包括一个“根框架”,它拥有 `QTextBlock` 和 `QTextFrame` 序列。这些框架可包含多个块(还有其他内容,如多个列表和表格)。每个块包含一个或更多的文本片段,每个文本片段都有自己统一的格式(在 9.1 节中讨论过 `QTextDocument` 结构;请参考图 9.1)。

既然我们只关心文本(它其实只是一个块,因为 `RichTextLineEdit` 只有一行),我们可以遍历 `QTextDocument` 的文本块,忽略框架——无论如何,在根框架之外应该就没有任何框架了(在这种特殊的情形下,我们甚至要丢掉外部循环,仅在第一个文本块处运行,因为只有一个文本块,但我们更倾向于使用一个更加普通的方法)。

一旦拥有了有效的文本块(在此处,也就是行)就必须遍历它所有的文本片段。实际上,如果行的文本格式都相同,将只有拥有一个格式的一个片段。在此处的“bear”示例中,行已经有几个不同的格式(例如,一些粗体文本和一些其他颜色的文本),因此我们预计一个单独的文本块,它拥有 5 个片段:“The”(包括紧随的空格)、“bold”(粗体)、“ ”(一个空格)、“blue”(蓝色)和“bear”(包括前面的空格)。

提取格式的每一个片段(存储在 `QTextCharFormat` 对象中)及其颜色和文本。将文本 HTML 化,也就是把“&”、“<”和“>”转换成等效的 HTML 符号(&、<和 >)。然后基于片段的格式建立一个 HTML 格式标志的字符串列表。此处需要一个列表,因为将可能设定多格式属性,如粗体和斜体。只要有了标志列表,把每一项封装成开始-结束标志(start-end tag),此标志包含在文本周围(还有前面任何标志对的周围),如果文本的颜色不是黑色,把文本封装在一对 标志中,以设定它的颜色。然后添加文本到我们建立的 html 字符串,添加的文本现在已拥有 HTML 需要的所有格式,html 字符串由最后的方法返回。

到此为止,我们已完成了对 RichTextLineEdit 的介绍。使用这个类或由它所激发出的类可以给予用户输入单行富文本的方法,或许还可以把他们的文本存储到最紧凑的 HTML 格式中。然而,出乎意料的是 Qt 已经通过 QTextEdit 类支持了多行富文本编辑。但为了让 QTextEdit 更有用于最终用户,我们需要提供一些方法,例如,应用字体效果等,这和为 RichTextLineEdit 所做的处理一样。下一节将介绍这些内容。

9.4 编辑多行的富文本

Qt 的 QTextEdit 类提供了很多功能,我们就不必再去做很多工作,以试图把它扩展成一个有用的富文本编辑器。本节中,我们将创建 TextEdit 类,如图 9.6 所示。这个类把 QTextEdit 几个工具条结合了起来,实现了触发粗体和斜体开关、设定一个文本颜色、字体、字体尺寸和文本的对齐。这些绝不是我们所能支持的所有字符和段落格式,但足以用来解释说明相关的原理和实践过程。Text Edit 示例(textedit)中用 TextEdit 类提供测试和实验;第 12 章的 Page Designer 应用程序使用了 TextEdit,但关闭了它的对齐功能。

TextEdit 类必须要实现的基本目标有两个。首先,它必须为用户提供一种方法,要让用户可以应用想要的格式。其次,它必须给出光标位置处起作用的格式。

大部分工作都涉及到了便捷的 TextEdit 的创建:创建并准备窗口部件、创建动作、布局窗口部件和创建连接。以下是所有窗口部件的私有成员函数(从 textedit.hpp 中获取),仅提供部分内容:

```
QToolBar *fontToolBar;
QAction *boldAction;
QAction *italicAction;
QAction *colorAction;
QColorDialog *colorDialog;
QFontComboBox *fontComboBox;
QDoubleSpinBox *fontSizeSpinBox;
QToolBar *alignmentToolBar;
QAction *alignLeftAction;
QAction *alignCenterAction;
QAction *alignJustifyAction;
QAction *alignRightAction;
QTextEdit *textEdit;
```

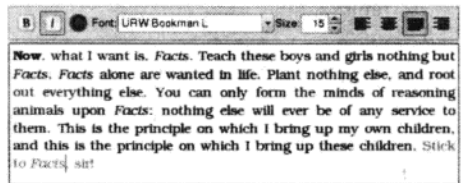


图 9.6 TextEdit 组件

对齐动作放在 createWidgets() 方法中的一个动作组中(在此未做介绍)。通过这样做,可以确保一个时间段内,只有一个对齐方式被选中,因为动作组默认它的这些动作是互斥的。

窗口部件的大部分行为可以重新实现,要通过调用聚集的 textEdit 完成,TextEdit 使用简单的适配器方法。以下为一些示例,同样是从头文件中提取而来:

```
QString toHtml() const { return textEdit->toHtml(); }
void setHtml(const QString &html) { textEdit->setHtml(html); }
void setBold(bool on)
{ textEdit->setFontWeight(on ? QFont::Bold : QFont::Normal); }
void setFontPointSize(double points)
{ textEdit->setFontPointSize(static_cast<qreal>(points)); }
void setFontFamily(const QFont &font)
{ textEdit->setFontFamily(font.family()); }
void alignLeft() { textEdit->setAlignment(Qt::AlignLeft); }
```

除了这些,还有其他适配器方法,也可以传递工作到聚集的 QTextEdit,如 toPlainText()、alignCenter() 等。我们没有对此进行介绍,因为它们都与这些介绍过的内容相类似。

一些动作(如触发斜体)可以仅通过信号和槽连接重新实现,而其他的则要求有一个适配器或一个自定义方法,但我们将跳过构造函数、createWidgets() 和 createLayout() 方法,因为它们都很普通。


```

void TextEdit::createConnections()
{
    connect(fontComboBox, SIGNAL(currentFontChanged(const QFont&)),
            this, SLOT(setFontFamily(const QFont&)));
    connect(fontSizeSpinBox, SIGNAL(valueChanged(double)),
            this, SLOT(setFontPointSize(double)));
    connect(boldAction, SIGNAL(toggled(bool)),
            this, SLOT(setBold(bool)));
    connect(italicAction, SIGNAL(toggled(bool)),
            this, SLOT(setFontItalic(bool)));
    connect(colorAction, SIGNAL(triggered()), this, SLOT(setColor()));
    connect(alignLeftAction, SIGNAL(triggered()),
            this, SLOT(alignLeft()));
    ...
    connect(textEdit, SIGNAL(currentCharFormatChanged(
            const QTextCharFormat&)),
            this, SLOT(currentCharFormatChanged(
            const QTextCharFormat&)));
    connect(textEdit, SIGNAL(cursorPositionChanged()),
            this, SLOT(cursorPositionChanged()));
    connect(textEdit, SIGNAL(textChanged()),
            this, SIGNAL(textChanged()));
}

```

此处给出的前5个连接都是关于改变字符属性的,第6个与改变段落属性相关。我们仅通过使用 QFontComboBox 的 currentFontChanged() 来改变字形库。字体尺寸通过一个来自 fontSizeSpinBox 的连接改变。粗体和斜体动作都通过触发(可选项)动作来触发相应的字体属性。颜色动作用来弹出一个颜色选择对话框。除了左对齐触发动作的连接以外还有为其他对齐方式(中央对齐、强制对齐、右对齐)而设的等效连接,这些都未做介绍。

注意,我们可以直接把斜体动作连接到聚集的 QTextEdit,但除了颜色动作之外,所有其他动作都连接到小的适配器方法,这些适配器方法在头文件中(前面给出)执行,它们依次调用合适的 QTextEdit 方法。

倒数第二的两个连接用来确保工具条的窗口部件正确地反映了当前光标位置处字符和段落的状态。

段落状态可能会因为导航(用户点击文本的其他地方,或使用方向键或其他键移动到其他地方)的变化而改变或因为当前位置的变化而改变。例如,用户点击了粗体的工具条按钮。最后的连接(一个信号-槽连接)用来提供给 TextEdit 一个信号以匹配聚集的 QTextEdit 的信号。

我们将介绍两个方法,它们可以让工具条按钮的状态保持最新,但首先要看一下 setColor() 方法和它的帮助槽,因为它是唯一的改变状态的方法,此方法不能在头文件中用一个适配器重新实现。

```

void TextEdit::setColor()
{
    if (!colorDialog) {
        colorDialog = new QColorDialog(this);
        connect(colorDialog, SIGNAL(colorSelected(const QColor&)),
                this, SLOT(updateColor(const QColor&)));
    }
    colorDialog->setCurrentColor(textEdit->textColor());
    colorDialog->open();
}

```

此方法弹出一个对话框,它是一个颜色选择器,窗口模式由平台定义,它的初始设定为当前文本颜色^①。如果用户点击了 OK 按钮,发射 colorSelected() 信号。我们已将它连到了一个自定义的 updateColor() 槽。

① 遗憾的是,此方法不能在 Mac OS X 下使用的 Qt 4.6 中正常工作。在作者的 MacBook 上运行的 Leopard (10.5.8) 系统下, Qt 4.6.0 出现程序错误,而对于 Qt 4.6.1 和 Qt 4.6.2,色彩设定一次正常,但第二次尝试时就会出现程序冻结。因此,Mac OS X 下使用的 Qt 4.6 中包含多个 #if 的示例就使用了静态的 QColorDialog::getColor() 方法。

```
void QTextEdit::updateColor(const QColor &color)
{
    textEdit->setTextColor(color);
    updateColorSwatch();
}
```

这个槽用来把当前文本颜色设定为给定颜色,并更新工具条中使用的颜色卡。工具条的颜色动作使用一个显示了一个粗体“C”的图标,它的背景是当前文本的前景颜色。无论何时颜色改变,都必须通过 `updateColorSwatch()` 方法更新图标。

```
void QTextEdit::updateColorSwatch()
{
    colorAction->setIcon(colorSwatch(textEdit->textColor(),
                                     QSize(48, 48)));
}
```

此方法调用了自定义全局 `colorSwatch()` 函数,此函数获取一个颜色和尺寸,并返回一个所请求尺寸和颜色的图标。第 12 章中的 Page Designer 应用程序使用了 `colorSwatch()` 和一些其他函数 (`brushSwatch()`、`penStyleSwatch()`、`penCapSwatch()` 和 `penJoinSwatch()`)。

虽然代码只有一行,我们还是倾向于把它放在一个自定义方法中,因为不仅只有一个地方需要它的功能。

```
void QTextEdit::currentCharFormatChanged(
    const QTextCharFormat &format)
{
    fontComboBox->setCurrentFont(format.font());
    fontSizeSpinBox->setValue(format.fontPointSize());
    boldAction->setChecked(format.fontWeight() == QFont::Bold);
    italicAction->setChecked(format.fontItalic());
    updateColorSwatch();
}
```

无论何时,只要当前字符格式改变,就调用这个槽——它不是为段落级的改变调用的,如改变段落的对齐。它简单地更新工具条的窗口部件,以反映当前字符格式。格式改变的原因是状态发生了改变(如用户改变了颜色),也可能是因为导航发生了改变(用户移动到一个拥有不同格式的字符,这个字符与光标所在的前一个字符的格式不同)。

```
void QTextEdit::cursorPositionChanged()
{
    QTextCursor cursor = textEdit->textCursor();
    QTextBlockFormat format = cursor.blockFormat();
    switch (format.alignment()) {
        case Qt::AlignLeft:
            alignLeftAction->setChecked(true); break;
        case Qt::AlignCenter:
            alignCenterAction->setChecked(true); break;
        case Qt::AlignJustify:
            alignJustifyAction->setChecked(true); break;
        case Qt::AlignRight:
            alignRightAction->setChecked(true); break;
    }
}
```

只有在想追踪段落格式改变的时候,才需要这个槽。无论何时,只要光标位置改变,就意味着当前光标在一个不同的段落中。因此,我们只允许用户改变段落的对齐,这也是唯一需要对格式进行追踪的部分。此处,我们检查哪一个对齐动作匹配段落的对齐,依赖于互斥的动作组,它不选择所有其他项。

对于 `TextEdit`,我们选择了最常见字符格式的一个子集和段落格式简单的对齐,用以介绍如何

提供给用户一些方法,让他们能改变这些格式,还有,如何在用户界面上反应当前状态。还有其他一些我们需要提供的功能、用户的控制方法和我们的追踪方法。例如, `QTextCharFormat` 拥有控制上画线、删除线和下画线(包括线的颜色和样式)的方法,还有设定一个 URL 和工具提示的方法。正如我们前面讨论富文本单行编辑器时看到的那样,它还支持垂直定位以产生下标和上标。还有许多其他我们可以为段落提供的功能。例如, `QTextBlockFormat` 拥有设定一个段落第一行缩进以及所有缩进的方法,还有设定边距和制表符的方法。

如果想更进一步,还可以提供用户创建列表和表格,用它们来插入超链接和图像的功能。实际上,为项目符号列表提供基本支持是极其简单的:只要调用 `QTextEdit::setAutoFormatting(QTextEdit::AutoBulletList)`,用户就可以通过在左页边处插入一个“*” (星号),这样就开始了个项目符号列表。然而,要提供缩进和不缩进(到嵌入的列表),插入和编辑表格,等等,则需要更多的工作要完成。所有这些事情都留给那些想对 Qt 的富文本引擎做深入开发的人作为练习。

到此,我们已经完成了对创建富文本编辑器的介绍,包括如何提供输入提示和语法加亮。但如果要有计划地创建富文本文档——例如,产生工资条或月底用户的账单,该如何做呢?如何打印这样的文件或把它们输出到标准文本格式呢?我们将在下一章阐述这些问题,并给出如何简单地赋色,而完全不需要创建一个 `QTextDocument`。



第 10 章 创建富文本文档

- 高质量地输出 QTextDocument 文件
- 创建 QTextDocument
- 输出和打印文档
- 绘制页面

在这一章,我们将会看到三种不同的创建富文本文档的方法,也会看到如何以不同的标准格式来输出这些文档,还会看到如何打印它们^①。本章会认为读者已对 QTextDocument 有了基本了解,就像 9.1 节里介绍的那样。对于这三种将要用到的方法,第一种方法是用原生的 HTML 填充 QTextDocument,第二种方法是用 QTextCursor 填充 QTextDocument,第三种方法是用 QPainter 绘制文档。对于这里的最后一种方法仅是为了输出和打印,没有用到内存中表示(in-memory representation)。

遗憾的是,这可能是 Qt 编程中一个令人失望的地方,特别是对那些期望跨平台的解决方案。这是因为 Qt 在对富文本文档输出和打印的行为会因跨越的平台和 Qt 的版本不同而有很大的不同。有鉴于此,我们将会针对特定的 Qt 版本,Qt 4.5.2 和 Qt 4.6.1,针对 Mac OS X、Linux 和 Windows 平台,给出一个总结,让用户可以容易地根据自己的环境获取最佳的表现。然而,需要注意的是,将要处理的示例文档相当复杂(它包括一个表格、一些 SVG 嵌入式图片,还有一些富文本),因此,对于其他文档,或者甚至是根本不同的文档来说,所得的结果也可能不会有太多不同。

我们将要生成的双页示例文档如图 10.1 所示。该图是一个 PDF 文件的截图,这个文件是在 Linux 系统下使用 Qt 4.5.2 从原生的 HTML 创建的 QTextDocument 中输出而来的。

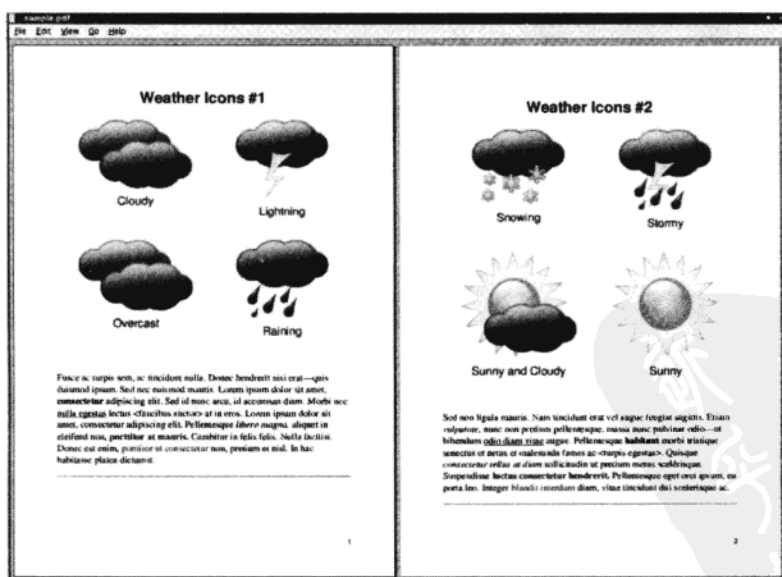


图 10.1 两页富文本示例文档在 evince 中的浏览效果

① 如前一章所述,Qt 的富文本格式是一个常驻内存的数据格式,不要和其他的“富文本”格式相混淆。

在这一章,我们会先通过对这些输出文件的质量和大小比较开始——这些文件只是该示例文档的不同格式。在此之后会介绍 `QTextDocument` 的用法,在 10.2 节会介绍用它来填充原生的 HTML 和 `QTextCursor` 的用法。10.3 节将会看到如何用不同的格式来输出 `QTextDocument`,也会看到如何来打印 `QTextDocument`。而在本章的最后一节将会看到如何只通过绘制方式来创建这同一文档,还会看到如何输出或打印这些结果。

在深入介绍本章的各节内容之前,我们会快速查看一下示例文档页面数据源的数据结构,因为这对涉及到的所有方法来说都是通用的。

```
struct OnePage
{
    QString title;
    QStringList filenames;
    QStringList captions;
    QString descriptionHtml;
};
```

`title` 包含的是纯文本的页面标题。这两个字符串列表可分别用于图片文件名和纯文本标题文字的设置。`descriptionHtml` 保存的是表格之后的文本段落,对于文本效果(如颜色、粗体,等等)它用到的是 HTML 标记(markup)。

用于文档的数据保存在自定义的 `PageData` 对象中,它包含一个类型为 `QList < OnePage >` 的私有变量 `pages`。我们不会介绍这些驱动类(driver class),甚至不会介绍 `PageData` 类的太多内容;相反,我们会把重点放在必须用来说明如何填充 `QTextDocument` 和如何输出或打印文档的那些方法上。

10.1 高质量地输出 `QTextDocument` 文件

在这一节,我们将会对示例文档以 Qt 支持的多种标准格式输出(和绘制)文件的质量和大小进行比较。

后续的这些图片设计用于给出每种方法的预期结果^①。但由于跨越的格式、平台、Qt 版本和文档之间的差异,建议读者对这里给出的结果要保有怀疑精神,并且要亲自测试一下。在本章中给出的所有输出都是由图 10.2 所示的 Output Sampler 示例程序(output sampler)生成的。仅用于测试目的,修改这个示例程序来输出自己的文档,应该还是比较简单的。

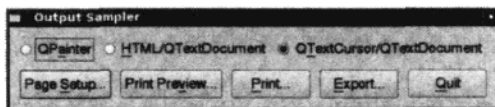


图 10.2 Output Sampler 程序

图 10.3 给出了通过 `QPrinter` 导出和输出到文件(output to file)的 PDF 输出的比较。PDF 文件也可以通过打印对话框来打印文档,选择“打印到文件”(print to file)选项——这通常会产生一个与输出方法不同的 PDF 输出,这种输出结果可能会稍好一些,也可能会稍差一些^②。另外一个需要注意的问题是,`QTextDocument` 会把页码放在每一页的右下角,而不考虑我们是否想要这种结果,并且也无法控制页码的格式。当然,对于由 `QPainter` 产生的文档,我们需要自己绘制每一样东西。

① “quality”的星数反映了作者的主观意见,它会因文档的不同而有所差异。字节的大小也会发生变化——它们仅用做相关文件大小的指示。

② 在 Windows 中,如果安装了 PDF 打印机驱动程序,“打印到文件”(print to file)才会生成 PDF 文件。如果没有安装 PDF 打印机驱动程序,在 Windows 下就只能通过输出来产生 PDF 文件。

Platform/ Qt version	PDF Output Technique (Quality Bytes)		
	QTextDocument/ HTML	QTextDocument/ QTextCursor	QPainter
Linux/4.5.2	***** 136 251	***** 136 216	***** 5 956 141
Linux/4.6.1	***** 143 985	***** 136 302	***** 6 669 547
Mac/4.5.2	***** 135 407	***** 135 316	***** 5 991 845
Mac/4.6.1	***** 135 890	***** 135 481	***** 6 687 444
Windows/4.5.2	***** 147 838	***** 150 680	***** 5 998 190
Windows/4.6.1	***** 148 045	***** 150 881	***** 6 689 564

图 10.3 PDF 输出质量和大小的比较

图中显示的文件大小数字表明,对有 SVG 嵌入图片的 PDF 来说,使用 QPainter 输出会产生很大的文件。如果输出的是像素图片(这可以通过取消 pro 文件中的 EMBED_SVG 定义来实现),QPainter 产生的 PDF 大小将会减少到 100 KB 左右。

Outputsampler 程序还可以用 PostScript(.ps)格式输出文档。在我们的测试中(在 Linux 平台上使用 Qt 4.6.1),我们发现示例文档的 Qt PostScript 输出大小会有显著差异,这取决于使用的是 QTextDocument(带 SVG 嵌入图片的大约为 7 MB,带.png 嵌入图片的大约为 170 KB)还是 QPainter(无论嵌入的是哪一种图片格式,都大约是 16 MB)。同时,我们还发现,示例文档的 PostScript 文件无法用 evince 浏览器(projects.gnome.org/evince)查看,但在更为基础的 gv 浏览器(www.gnu.org/software/gv)上却没有问题。不管怎样,我们不会提供一个加以比较的表格,因为 PDF 格式的绝大多数效果都要胜过 PostScript。

图 10.4 给出了示例文档输出为开放文档格式(Open Document Format(.odt))文件的质量和大小对比结果。这些结果只能用 QTextDocument 输出,它们无法用 QPainter 绘制完成。我们用 OpenOffice.org 3(www.openoffice.org)^①对输出的 .odt 文件进行了测试。图中数字清晰地表明 Qt 对这种格式的支持还不太成熟,也还在不断改进中。

虽然在我们的测试中 Qt 的表现差强人意,但因此而舍弃 Qt 能够输出良好的 .odt 文件的能力就显得不够明智了。我们使用的示例文档虽然有点短,但却相当复杂,使用其他形式的文档可能会获得较好的结果。当然,Qt 输出 .odt 格式的能力肯定会在未来的版本中得到进一步的改善。

图 10.5 给出了 HTML 格式输出结果的比较。HTML 是一种通用的格式,因而许多应用程序都必须能够输出 HTML。Qt 的 XML 类让这件事变得容易起来(例如,可以使用 QXmlStreamWriter),但我们通常只希望得到一个单独的内存中的文档(如 QTextDocument),这样就可以从这个文档中产生一系列的输出格式,其中之一就是 HTML。正如开放文档格式一样,HTML 也不能用 QPainter 进行绘制。

在 Mac OS X 上,我们使用 Safari 网络浏览器进行测试——它可以非常好地绘制 HTML 页面,包括 SVG 嵌入图片。在 Linux 和 Windows 上,我们使用 Firefox 进行测试,它却根本无法显示 SVG 图片。

图 10.6 给出了可缩放矢量图(Scalable Vector Graphics(.svg))格式输出结果的比较——在这

^① 专业术语是“Open Document Format”,它包括面向富文本文档(.odt)、面向电子表格(.ods)和其他文档类型的格式。使用文件名后缀 .odt(Open Document Text)是为了让 OpenOffice.org 正确地予以识别。

个示例中,我们仅输出了示例文档的第一页。SVG 支持不降低图像质量的缩放——这正如其名字所隐含的意思一样。

Platform/ Qt version	ODT Output Technique (Quality Bytes)	
	QTextDocument/ HTML	QTextDocument/ QTextCursor
Linux/Qt 4.5.2	☆☆☆☆ 135 432	☆☆☆☆ <i>invalid</i>
Linux/Qt 4.6.1	☆☆☆☆ 135 437	☆☆☆☆ 135 438
Mac/Qt 4.5.2	☆☆☆☆ 135 441	☆☆☆☆ <i>invalid</i>
Mac/Qt 4.6.1	☆☆☆☆ 135 444	☆☆☆☆ 135 443
Windows/Qt 4.5.2	☆☆☆☆ 135 407	☆☆☆☆ <i>invalid</i>
Windows/Qt 4.6.1	☆☆☆☆ 135 413	☆☆☆☆ 135 409

图 10.4 开放文档格式输出质量和大小的比较

Platform/ Qt version	HTML Output Technique (Quality Bytes)	
	QTextDocument/ HTML	QTextDocument/ QTextCursor
Linux/Qt 4.5.2	★★★★☆ 8 355	★★★★☆ 6 795
Linux/Qt 4.6.1	★★★★☆ 8 346	★★★★☆ 6 795
Mac/Qt 4.5.2	★★★★★ 8 363	★★★★★ 6 803
Mac/Qt 4.6.1	★★★★★ 8 363	★★★★★ 6 803
Windows/Qt 4.5.2	★★★★☆ 8 106	★★★★☆ 6 539
Windows/Qt 4.6.1	★★★★☆ 8 106	★★★★☆ 6 539

图 10.5 HTML 输出质量和大小的比较

Platform/ Qt version	SVG Output Technique (Quality Bytes)		
	QTextDocument/ HTML	QTextDocument/ QTextCursor	QPainter
Linux/Qt 4.5.2	★★★★☆ 94 535	★★★★☆ 94 560	★★★★★ 94 156
Linux/Qt 4.6.1	★★★★☆ 91 756	★★★★☆ 91 436	★★★★☆ 102 061
Mac/Qt 4.5.2	★★★★☆ 91 203	★★★★☆ 91 195	★★★★☆ 94 124
Mac/Qt 4.6.1	★★★★☆ 92 905	★★★★☆ 91 804	★★★★☆ 99 360
Windows/Qt 4.5.2	★★★★☆ 92 066	★★★★☆ 92 087	★★★★★ 95 338
Windows/Qt 4.6.1	★★★★☆ 92 597	★★★★☆ 92 663	★★★★☆ 100 544

图 10.6 SVG 输出质量和文件大小的比较

还可以使用诸如. png(便携式网络图像, Portable Network Graphics) 或. bmp(Windows 位图格式, Bitmap) 的像素图片格式输出文档。这些格式不能很好地支持缩放(这是它们自身设计上所带来的缺陷, 与 Qt 无关)。因此, 这些格式对于那些无须缩放的图像来说是最理想的选择。我们没有提供这两种格式的对比表格, 因为对于所有平台、Qt 版本和所用到的其他格式的技术中, 这两种格式的输出都具有极好的保真度。

10.2 创建 QTextDocument

有两种技术可用于填充 QTextDocument: 给它一个 HTML 格式的 QString 文本字符串, 或者使用 QTextCursor。我们将会在这一节介绍这两种方法, 先从 HTML 的用法开始。

当然, 某些情况下根本不需要绘制一个文档, 而只想以一个标准格式进行绘制, 或是打印数据。这时(假定我们不需要 HTML 或者 ODF 输出) 完全可以不用 QTextDocument, 而仅用 QPainter 就可以了; 这一点会在本章的最后一节中介绍。

10.2.1 使用 HTML 创建 QTextDocument

如果你已有相当好的 HTML 知识, 那么使用 HTML 填充 QTextDocument 会显得非常方便, 这也可能是本章介绍的所有方法中需要代码量最少的一种方法。

在 Outputsampler 示例中, 要使用 HTML 填充 QTextDocument, 先从创建一个空 QTextDocument

开始,然后向 `PageData::populateDocumentUsingHtml()` 方法传递一个指向该文档的指针,这个方法自己还有两个帮助方法(helper method)。

```
void PageData::populateDocumentUsingHtml(QTextDocument *document)
{
    QString html("<html>\n<body>\n");
    for (int page = 0; page < pages.count(); ++page) {
        html += pageAsHtml(page, false);
        if (page + 1 < pages.count())
            html += "<br style='page-break-after:always;' />\n";
    }
    html += "</body>\n</html>\n";
    document->setHtml(html);
}
```

这个方法会创建一个 `QString` 的 HTML,然后会对传递的 `QTextDocument` 进行设置。这是通过添加一个标准 HTML 的 `<html>` 标签和省略 `<head>` 标签开始的,还会在文档的开头部分添加一个适当的 `<body>` 标签。然后,对 `OnePage` 列表中的每一页添加 HTML,并对除最后一页外的所有其他页添加一个 Qt 特有的 `page-break-after` 样式属性标签,它会让页面强制分页。最后,会封闭所有的开始标签。

需要注意的是,之前选择把换行符(`\n`)包括在我们的 HTML 代码中;但并不必要这样做,但在某些时候,这样做很方便调试,并且可以让 HTML 更具人工阅读性。

```
QString PageData::pageAsHtml(int page, bool selfContained)
{
    const OnePage &thePage = pages.at(page);

    QString html;
    if (selfContained)
        html += "<html>\n<body>\n";
    html += QString("<h1 align='center'>%1</h1>\n")
        .arg(Qt::escape(thePage.title));
    html += "<p>";
    html += itemsAsHtmlTable(thePage);
    html += "</p>\n";
    html += QString("<p style='font-size:15pt;font-family:times'>"
        "%1</p><hr>\n").arg(thePage.descriptionHtml);
    if (selfContained)
        html += "</body>\n</html>\n";
    return html;
}
```

如果第二个参数的值为 `true`,这个方法会产生一个简单的自包含 HTML 页面;而 `populateDocumentUsingHtml()` 方法总是通过设定这个参数值为 `false` 来调用它。

这个方法在开始时,会先获取一个要转换为 HTML 页面数据的引用。它会给标题创建一个 `<h1>` 标签,并且要小心使用 Qt 的 `Qt::escape()` 函数来把标题中任意的“&”、“<”和“>”字符转换成 `&`、`<` 和 `>`;(因为这些字符是 HTML 文件中的实体字符)^①。

用于图片和图片标题的 HTML 表格会由一个单独的方法进行处理,接下来将会看到它。`descriptionHtml` 文本会不经变换(unescaped)地添加到最后(因为它已经是 HTML 格式了),但在它之前还会放置一个段落标签,这个样式属性会提供适当的字体库和字体大小。在文本的后面还会添加一条水平线。

在图 10.1 中给出的屏幕截图的页码是由 Qt 添加的,我们无法控制它。

① 在 Qt 4.5 中, `Qt::escape()` 并不适用于为 HTML 属性值转换文本,因为它不会转换引号。从 Qt 4.6 开始, `Qt::escape()` 就已经能转换双引号了。


```

QString PageData::itemsAsHtmlTable(const OnePage &thePage)
{
    QString html("<table border='1' cellpadding='20' width='100%'>");
    for (int i = 0; i < thePage.fileNames.count(); ++i) {
        if (i % 2 == 0)
            html += "<tr>\n";
        html += QString("<td align='center'><img src='%1' />"
            "<p style='font-size:18pt'>%2</p></td>\n")
            .arg(thePage.fileNames.at(i))
            .arg(Qt::escape(thePage.captions.at(i)));
        if (i % 2 != 0)
            html += "</tr>\n";
    }
    if (!html.endsWith("</tr>\n"))
        html += "</tr>\n";
    html += "</table>\n";
    return html;
}

```

这个方法会创建一个带有 4 个单元格 (2 × 2) 的 HTML 表格。每个单元格都包含一个 标签, 它的 src 属性是图片文件名, 还包含一个段落标签, 这个标签以大号字显示相应的经 HTML 转换后的纯文本标题 (默认情况下, outputsampler 程序使用 SVG 图片, 但也可以通过注释 .pro 文件中的 EMBED_SVG 定义符号而让它使用 PNG 图片)。

这三个方法就足以创建一个包含 HTML 文本的长 QString 来显示文档。用 HTML 字符串填充 QTextDocument 只需简单调用 QTextDocument::setHtml() 方法。尽管没有像上一章提到的那样在这个示例中使用 CSS (Cascading Style Sheets, 层叠样式表), 但 Qt 的确还是支持这种方法的; 具体内容可以参阅 qt.nokia.com/doc/richtext-html-subset.html。

10.2.2 使用 QTextCursor 创建 QTextDocument

QTextCursor 类允许我们通过编程的方法来浏览和编辑 QTextDocument 而无须了解任何 HTML 知识。QTextCursor 类曾在上一章中介绍过, 当时它用于文档的编辑; 在本小节中, 我们将简单地用它来创建 QTextDocument。在前一章中, 我们已经介绍了 QTextCursor 的 API。也在前一章介绍了 QTextCursor::movePosition() 方法, 同时还介绍了 QTextCursor::MoveOperation 的枚举值。

QTextCursor API 的不同寻常之处在于, 它有两种不同的方法来插入列表: createList() 和 insertList()。前者会创建文档, 向该文档插入一个列表, 并以当前段作为列表的第一项。后者会创建文档, 向该文档插入一个列表, 并创建一个新的段落来作为列表的第一项。

使用 QTextCursor 填充 QTextDocument 的代码与用来创建 HTML 字符串的代码在结构上非常相似, 但需要用到更多的帮助方法。对于 HTML 版本, 先从创建一个空 QTextDocument 并向它传递一个指针开始, 该指针会指向一个填充文档的方法。

```

void PageData::populateDocumentUsingQTextCursor(
    QTextDocument *document)
{
    document->setDefaultFont(QFont("Times", 15));
    QTextCursor cursor(document);
    for (int page = 0; page < pages.count(); ++page) {
        addPageToDocument(&cursor, page);
        if (page + 1 < pages.count()) {
            QTextBlockFormat blockFormat;
            blockFormat.setPageBreakPolicy(

```



```

QTextDocument tableDocument;
QTextCursor tableCursor(&tableDocument);
QTextTable *table = tableCursor.insertTable(2, 2, tableFormat());
for (int i = 0; i < thePage.fileNames.count(); ++i)
    populateTableCell(table->cellAt(i / 2, i % 2), thePage, i);
cursor->insertFragment(QTextDocumentFragment(&tableDocument));
}

```

通过编程来用 QTextCursor 向 QTextDocument 添加一个表格的过程稍显啰唆,尽管没有哪一步是有困难的。

如果试图用 QTextCursor::insertTable() 方法把 QTextTable 直接插入到 QTextDocument 中,将很容易使事情变得一团糟! 因为我们必须记住,一旦填充了表格,就必须在填充文档其他内容之前立刻把光标移到表格的后面。说到这一问题的原因是在把最后一项插入到表格中最后一个单元格时,光标会立即定位到插入项的后面——但它仍然会在表格的最后一个单元格里。

幸运的是,这一问题有一个非常好的通用解决方案:创建一个独立的 QTextDocument,它只包含表格,然后把这个文档当做文档片段插入到想要填充的文档中。这样做会让光标整齐地定位在片段的后面(也就是紧贴着表格的后面),因此也就可以继续填充文档而不必总是担心要明确地把光标移出表格。

因此,在这个方法中,我们创建了一个新的 QTextDocument,并创建了一个用来填充这个 tableDocument 的新光标。先从调用 QTextCursor::insertTable() 开始,给定它的行数和列数,还有要使用的 QTextTableFormat(接下来将会看到 tableFormat() 方法)。

一旦拥有 QTextTable,就可以用帮助方法来填充它的每个 QTextTableCell 了——可以用 QTextTable::cellAt() 方法检索。

最后,把表格文档当做 QTextDocumentFragment 插入到正在填充的文档中。这时,在插入内容后指针会立即得到定位(就像平时在指针后插入内容一样)。这意味着指针会精确地定位到我们所希望的位置:紧跟在表格的后面。

```

QTextTableFormat PageData::tableFormat()
{
    QTextTableFormat tableFormat;
    tableFormat.setAlignment(Qt::AlignCenter);
    tableFormat.setCellPadding(10);
    tableFormat.setTopMargin(10);
    tableFormat.setBottomMargin(10);
    QVector<QTextLength> widths;
    widths << QTextLength(QTextLength::PercentageLength, 50)
           << QTextLength(QTextLength::PercentageLength, 50);
    tableFormat.setColumnWidthConstraints(widths);
    return tableFormat;
}

```

在这个方法中我们创建了 QTextTableFormat,设定了它的对齐设置、内填充和两个页边距的值(对于左右两个页边距使用默认值,因为我们没有对它们进行特别设置)。不是明确地设置这两列的宽度值,而是把每列的宽度值设置为表格总宽度的 50%。这样就可以把详细的计算工作留给 Qt 完成并确保两列都具有相同的宽度。

QTextLength 类还可以接受另外两个枚举值,VariableLength 和 FixedLength;我们自然可以为表格的列使用任意的长度组合值。每个 FixedLength 的值都是一个浮点型(qreal)的像素数。使用默认(无参数)的构造函数可以创建一个表示尺寸的 QTextLength 变量。

```

void PageData::populateTableCell(QTextTableCell tableCell,
                                const OnePage &thePage, int index)
{
    QTextBlockFormat blockFormat;
    blockFormat.setAlignment(Qt::AlignHCenter);
    QTextCursor cursor = tableCell.firstCursorPosition();
}

```

```
cursor.insertBlock(blockFormat);
cursor.insertImage(thePage_filenames.at(index));
blockFormat.setTopMargin(30);
cursor.insertBlock(blockFormat);
QTextCharFormat charFormat;
charFormat.setFont(QFont("Helvetica", 18));
cursor.insertText(thePage_captions.at(index), charFormat);
}
```

这个方法用来把图片和标题插入到每个传递给它的 `QTextTableCell` 中。由于我们想让图片和标题都水平居中,可以先从分别创建一个文本块格式和设置它的对齐方式开始。然后使用 `QTextTableCell::firstCursorPosition()` 方法在单元格中检索光标,并使用此光标插入一个拥有我们创建的块格式的空段落。然后再使用 `QTextCursor::insertImage()` 方法把图片插入到这个段落中。

`QTextCursor::insertImage()` 方法可以有多种不同方式的重载;我们在这里使用的重载带一个文件名,但还有一个重载可以接受一个 `QImage`。

为了确保图片底端和标题顶端之间有一小点垂直距离,我们下一步要设置块格式的顶部边距——这只影响该格式的后续部分。然后插入一个空段落,并再次用到该块格式。然后再创建一个字符格式并设置一个大号字,并使用该字符格式插入标题。

```
void PageData::addRuleToDocument(QTextCursor *cursor)
{
    QTextBlockFormat blockFormat;
    blockFormat.setProperty(
        QTextFormat::BlockTrailingHorizontalRulerWidth, 1);
    cursor->insertBlock(blockFormat);
}
```

在每一页的最后我们希望画一条水平线。通过创建一个文本块格式,把文本块格式的 `BlockTrailingHorizontalRulerWidth` 属性值设置为 1 并插入块的方式就可以很容易地做到这一点(实际上,在我们的测试中,把属性值设置为任何数字都可以绘制出水平线)。

`QTextFormat` 类(`QTextBlockFormat` 的基类)支持的属性超过 70 种(其中很多可以通过使用与属性相关的方法便捷地进行访问)。然而,`QTextFormat::setProperty()` 和各种各样的获取函数(例如,`QTextFormat::intProperty()`)的存在意味着属性的个数可以随时得到扩充(甚至是在很小的改动和补丁发布中),而不会影响其二进制兼容性。

10.3 输出和打印文档

`QTextDocument` 能够以不同的标准格式进行输出,包括开放文档格式和 HTML。这些文档也可以输出为矢量格式(如 PDF、PostScript 或 SVG)和 Qt 支持的任意像素图片格式(如 .png 或者 .bmp)。文档同样也可以进行打印。我们将在本节介绍所有这些内容。

10.3.1 输出 QTextDocument

在这一小节中,我们将会看到如何输出完全格式化的 `QTextDocument` 的内容,包括富文本和嵌入的图片。对于各种图片格式(SVG 和像素图片格式),我们将会看到如何输出单页上的单个图片,对于所有其他格式,我们将介绍如何输出完整的多页文档。

在所有情况下,我们都从创建一个空 `QTextDocument` 开始,对于单页 SVG 和像素图片输出,明确设定页面尺寸和边距。Qt 的文档通常不定义单位,一般来说,除用于打印外的单位都是像素(pixel)——用于打印的单位通常都是磅值(point)。

```
QTextDocument document;
document.setPageSize(printer.pageRect().size());
document.setDocumentMargin(25);
```

对于单文档,我们设定一个默认字体,创建一个 QTextCursor,把 QTextDocument 作为参数进行传递。然后使用前面介绍的 PageData::addPageToDocument() 方法向文档只添加一页。

对于多页文档,可以调用 populateDocument() 方法,它带一个指向空 QTextDocument 的指针参数。我们已经见过这个方法两个版本了:PageData::populateDocumentUsingHtml() 和 PageData::populateDocumentUsingQTextCursor()。理论上,我们使用的这两种方法不存在任何差异,但在我们的试验中发现这些结果通常会有所不同,因此建议读者对自己感兴趣的文档、平台和格式做一下测试。

10.3.1.1 输出 PDF 和 PostScript 格式

要把一个文档以 PDF 或 PostScript 格式输出是非常容易的,只需给定一个文件名和一个 QTextDocument。在这里给出的方法会假定接收到的文件名的后缀是 .pdf 或 .ps。

```
bool MainWindow::exportPdfOrPs(const QString &filename,
                               QTextDocument *document)
{
    Q_ASSERT(filename.endsWith(".ps") || filename.endsWith(".pdf"));
    QPrinter printer(QPrinter::HighResolution);
    printer.setOutputFileName(filename);
    printer.setOutputFormat(filename.endsWith(".pdf")
        ? QPrinter::PdfFormat : QPrinter::PostScriptFormat);
    document->print(&printer);
    return true;
}
```

要以 PDF 或 PostScript 格式输出,我们需要使用 QPrinter,但我们选择使用“打印到文件”(print to file)而不是打印到纸质页面^①(也可以使用 QPainter 向 QPrinter 绘制 PDF 和 PostScript 文档,后面将会看到这些内容)。

在大部分应用程序中,我们都应当把 QPrinter 作为主窗口中的私有成员。这样做的好处在于,一旦用户安装了打印机,后面的使用都会延续上次的用户设定。但对于 outputsampler 应用程序来说,我们选择为这些文档创建一个新的 QPrinter,这样在输出时就不需要用户的干预了——比如在输出示例文档的时候。

除了建立 QPrinter 有点麻烦,实际的输出过程仅是调用 QTextDocument::print(), 给它一个指向我们想要让文档打印的 QPrinter 指针。默认情况下,print() 方法会使用打印机的 QPrinter::paperRect() 作为自己的打印区域来给页码编数,但会留一个 20 mm 的页边距。

10.3.1.2 输出开放文档格式

以开放文档格式输出 QTextDocument 甚至要比以 PDF 或 PostScript 格式输出更容易,这要归因于 QTextDocumentWriter 类。

```
bool MainWindow::exportOdf(const QString &filename,
                           QTextDocument *document)
{
    Q_ASSERT(filename.endsWith(".odt"));
    QTextDocumentWriter writer(filename);
    return writer.write(document);
}
```

^① 在 Mac OS X 上,如果使用 QPrinter::NativeFormat 的一个输出格式,结果将是 Apple's Quartz 2D 绘画引擎渲染的 PDF。对于示例文件,这样做将会使 PDF 文件的大小翻倍,但产生文件的质量会稍有提升。

我们又一次传递了一个文件名(这次假设其后缀为 .odt) 和一个 QTextDocument。实际上, QTextDocumentWriter::write() 方法会向调用者返回一个布尔型的表示成功与否的标志。通过调用带有纯文本或 html 的 QByteArray 参数的 QTextDocumentWriter::setFormat() 方法, QTextDocumentWriter 也可以用来输出其他格式。默认的格式是“odf”, 但正如之前所提到的那样, 为了让 OpenOffice.org 能够识别文档, 文件名的后缀必须是 .odt。可以通过 QTextDocumentWriter::supportedFormats() 返回完整的格式清单(通常至少会包括这三种)。

遗憾的是, 如图 10.4 所示, Qt 的开放文档格式的输出功能(在撰写这部分内容的时候)还相当弱。我们的试验已经表明, 开放文档格式的输出效果对于文档的内容, 一些时候甚至对文档的填充方式都很敏感。有鉴于此, 推荐读者使用自己的 Qt 版本和目标平台输出自己的文档来测试输出质量, 以确保得到令你满意的结果。

10.3.1.3 输出 HTML 格式

为了以 HTML 格式进行输出, 我们必须创建一个包含 HTML 的字符串, 然后把字符串写入文件。这些都是标准的 C++/Qt 操作, 尽管在看完这些代码之后我们会提到一个简单的问题。

```
bool MainWindow::exportHtml(const QString &filename,
                           QTextDocument *document)
{
    Q_ASSERT(filename.endsWith(".htm") || filename.endsWith(".html"));
    QFile file(filename);
    if (!file.open(QIODevice::WriteOnly|QIODevice::Text)) {
        AQP::warning(this, tr("Error"),
                     tr("Failed to export %1: %2").arg(filename)
                     .arg(file.errorString()));
        return false;
    }
    QTextStream out(&file);
    out.setCodec("utf-8");
    out << document->toHtml("utf-8");
    file.close();
    return true;
}
```

Qt 的 QTextStream::setCodec() 方法对它所接受的编码名称十分宽松。例如, 我们这里就本应使用“utf8”。但 QTextDocument::toHtml() 方法则需要我们使用一个遵守 W3C (World Wide Web Consortium, 万维网联盟)^①的编码名称(如果完全给定编码)。如果没有给定编码, 那么 HTML 的元数据将不会包括 charset 属性; 对于 HTML 文件, 我们推荐总是使用 UTF-8 编码。

如果只需创建 HTML 文件, 那么就可以在代码中直接使用 QString 来完成(就像本章的前面部分, 在填充一个来自 HTML 的 QTextDocument 时所做的一样), 或者可以使用 Qt 的 QXmlStreamWriter, 它可以轻松保证属性和文本的正确转换(因为它已经为我们做到了)。

10.3.1.4 输出 SVG 格式

要以 SVG 格式输出我们的示例, 为简单起见, 只使用一个单独的页面来填充过 QTextDocument。执行输出的代码并不难, 但的确是包含了一些我们将要讨论的细节。需要注意的是, 要用 Qt 对 SVG 的支持, 必须在程序的 .pro 文件中添加 QT += svg 这行内容。

^① 关于 XML 的编码信息, 可以参阅 www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 www.iana.org/assignments/character-sets 中的内容。

```
bool MainWindow::exportSvg(const QString &filename,
                           QTextDocument *document)
{
    Q_ASSERT(filename.endsWith(".svg"));
    QSvgGenerator svg;
    svg.setFileName(filename);
    QRect rect = printer.paperRect().adjusted(25, 25, -25, 25);
    svg.setSize(rect.size());
    QPainter painter(&svg);
    painter.setViewport(rect);
    document->drawContents(&painter);
    return true;
}
```

通过向 `QSvgGenerator` 绘制可以产生一幅 SVG 图片。我们这里已使用了一个 `QPrinter` 成员变量，还需要把 SVG 图片的尺寸与打印机的页面尺寸相匹配，允许有一定的页边距。建立 SVG 发生器，创建一个 `QPainter` 来对它进行绘制，并把 SVG 图片的矩形区域设置成绘制器的视口（也就是说，将要实际绘制的区域）。`QTextDocument::drawContents()` 方法与 `QTextDocument::print()` 相似，只是它绘制的是绘制器而不是打印机，并且会接受一个可选的裁剪矩形区。

我们在这里已经使用 `QTextDocument` 便捷方法绘制了一幅 SVG 图片。然而，我们只能简单地使用 `QPainter` API 绘制图形、图片和文本来创建 SVG 图片，就像用我们所关心的任何其他设备绘图一样，在讲到 `paintSvg()` 方法时还会进一步看到这些内容。

10.3.1.5 输出像素图片格式

Qt 支持超乎想象的许多像素图片格式。这通常至少包括 `bmp` (Windows Bitmap, Windows 位图)、`jpg` 和 `jpeg` (Joint Photographic Experts Group, 联合图片专家组)、`png` (Portable Network Graphics, 便携式网络图形)、`ppm` (Portable Pixmap, 便携式像素图片)、`tiff` (Tagged Image File Format, 标签图片文件格式) 和 `xpm` (X11 Pixmap, X11 像素图片) 等，或许还会包括其他格式。准确的格式列表可通过 `QImageWriter::supportedImageFormats()` 予以提供（需要注意的是，Qt 可输出的格式通常比它可读的格式要少一些；可读的格式可以通过 `QImageReader::supportedImageFormats()` 予以返回。另外，还可以通过特定的格式插件来扩展格式的范围）。

```
bool MainWindow::exportImage(const QString &filename,
                             QTextDocument *document)
{
    QImage image(printer.paperRect().size(), QImage::Format_ARGB32);
    QPainter painter(&image);
    painter.setRenderHints(QPainter::Antialiasing|
                           QPainter::TextAntialiasing);
    painter.fillRect(painter.viewport(), Qt::white);
    painter.setViewport(printer.paperRect());
    document->drawContents(&painter);
    return image.save(filename);
}
```

在提供的文件后缀存在于支持的图片格式列表中时，无论使用何种文件格式表示的文件名后缀，这个方法都会把给定的 `QTextDocument` 输出到给定的文件中。

从结构上看，这里的代码与用于输出 SVG 图片的代码有些类似，但在这里我们会从创建一个 `QImage` 而不是 `QSvgGenerator` 开始。绘制像素图片到文件所用的代码与这些代码几乎相同，我们将会在随后介绍 `paintImage()` 方法时看到。

我们会创建一个绘制器来绘制图片，建立反走样并把图片的背景设置为白色（本可以简单地背景用 `Qt::transparent` 来代替这种做法）。按一般规律来说，在生成屏幕显示的图片 and 像素图片（如 `bmp` 和 `png`）时才会使用反走样，但在打印或制作矢量图（如 `svg`）时并不使用反走样。我

们让绘制器的视口与打印机的打印纸矩形范围相匹配(这一次没有页边距)并绘制文档的内容。最后,调用 `QImage::save()` (它会返回一个布尔型的表示成功与否的标志)来按照所需的格式把图片保存到文件中。

10.3.2 打印和预览 `QTextDocument`

`QTextDocument` 的打印与 `QTextDocument` 的输出非常相似,尽管通常会向用户提供设置打印机的选项,并且很有可能还需要提供在实际打印之前的页面预览功能。

对于 `outputsampler` 程序,我们已经选择把 `QPageSetupDialog` 指针作为主窗口的成员变量,并仅在需要的时候才会创建对话框。这使得向用户提供页面设置变得容易起来,并且除非实际用到对话框,否则几乎不耗用任何内存。

```
void MainWindow::pageSetup()
{
    if (!pageSetupDialog)
        pageSetupDialog = new QPageSetupDialog(&printer, this);
    pageSetupDialog->open();
}
```

如果没有提供页面预览,就可以向 `QPageSetupDialog::open()` 的调用传递参数,例如, `pageSetupDialog -> open(this, SLOT(print()))`; 如果用户接受了设置对话框,这样做就会导致 `print()` 槽的调用。使用 `open()` 而不是 `exec()` 是因为这样做可以让对话框在 Mac OS X 上显示成弹出窗口(sheet)^①; 在其他平台上,它则仍旧可以像平时那样显示成模式对话框(我们在之前也接触过这些内容——如第2章和第3章)。

对于打印预览,我们根据要求选择创建了一个 `QPrintPreviewDialog`——并用来在有请求发生时实现这一点。这看上去很有必要,因为预览对话框似乎会对其结果进行缓冲。这对 `outputsampler` 程序来说是不方便的,因为它会根据用户选择的是哪个单选按钮来决定页面的生成方式,所以我们每次都要强制刷新预览。在其他大部分应用程序中则没有必要这样做。

```
void MainWindow::printPreview()
{
    if (printPreviewDialog)
        delete printPreviewDialog;
    printPreviewDialog = new QPrintPreviewDialog(&printer, this);
    QSize size = QApplication->desktop()->availableGeometry().size();
    size.rwidth() /= 2;
    printPreviewDialog->resize(size);
    if (painterRadioButton->isChecked())
        connect(printPreviewDialog, SIGNAL(paintRequested(QPrinter*)),
                &pageData, SLOT(paintPages(QPrinter*)));
    else
        connect(printPreviewDialog, SIGNAL(paintRequested(QPrinter*)),
                this, SLOT(printDocument(QPrinter*)));
    printPreviewDialog->open();
}
```

我们希望对话框能够占据用户屏幕的整个高度,以使用户能够看到尽可能多的内容。通过 `QApplication::desktop()` 返回的 `QDesktopWidget` 可以为我们提供屏幕的实际几何尺寸(`screenGeometry()`),或者像这里的用法一样,对于可用的几何尺寸不能包括任务栏,在 Mac OS X 上还不能包括菜单栏和停靠栏(dock)。这两种方法都接受多头系统(multi-head system)的可选屏幕数,并且 `QDesktopWidget`

^① Mac OS X 的界面中所谓的“sheet”其实就是弹出窗口,但是它又与 Windows 的弹出窗口有所不同,因为这些弹出的窗口是“黏”在主窗口上的。试想一下,如果同时打开了多个窗口,且每个窗口都有一个弹出窗口,就很容易搞不清到底哪个弹出窗口是属于哪个窗口的了。相反,使用 sheet 就不会产生这样的问题——译者注。

`::screenCount()` 会返回现有的屏幕数量。由于现在大部分的桌面屏幕的宽度数值都比高度数值要大, 我们已把宽度设置成屏幕宽度的一半。

为了让打印预览有效, 必须提供一个连接到打印预览窗口 `paintRequested()` 的信号。对于 `outputsampler` 示例来说, 已经给两种可能性提供了信号, 尽管一般情况下只需要一种即可。第二个连接会连接到 `PageData::paintPages()` 槽(会在下一节介绍到)。

就像 `QPageSetupDialog` 的 `open()` 方法, 如果用户接受该对话框, 那么就会用一个接收对象和一个打印槽来调用 `QPrintPreviewDialog::open()` 方法。

图 10.7 给出了我们的打印预览测试结果。遗憾的是, 我们在 Mac OS X 上发现了问题: `QTextDocument` 的打印预览根本无法工作, 并且要绘制的文档在页面上的预览有些太窄了。

```
void MainWindow::printDocument(QPrinter *printer)
{
    QTextDocument document;
    populateDocument(&document);
    document.print(printer);
}
```

这个方法创建了一个空 `QTextDocument`, 用 `PageData::populateDocumentUsingHtml()` 或者 `PageData::populateDocumentUsingQTextCursor()` 来填充 `QTextDocument`, 然后会告诉文档在给定的打印机上打印。

```
void MainWindow::populateDocument(QTextDocument *document)
{
    Q_ASSERT(!painterRadioButton->isChecked());
    if (htmlRadioButton->isChecked())
        pageData.populateDocumentUsingHtml(document);
    else if (cursorRadioButton->isChecked())
        pageData.populateDocumentUsingQTextCursor(document);
}
```

出于完整性考虑给出了这个方法。它只是根据用户的选择来确定用哪一种填充文档的方法。

Platform/ Qt version	Print Preview Output Technique		
	QTextDocument/ HTML	QTextDocument/ QTextCursor	QPainter
Linux/4.5.2	✓	✓	✓
Linux/4.6.1	✓	✓	✓
Mac/4.5.2	✗	✗	✓
Mac/4.6.1	✗	✗	✓
Windows/4.5.2	✓	✓	✓
Windows/4.6.1	✓	✓	✓

图 10.7 打印预览的比较

10.4 绘制页面

通过创建一个 `QPainter`, 并使用 `PageData::paintPage()` 方法来绘制单文档(如 SVG 和像素图片文件)。对于多页文档, 方法几乎相同(也是先要创建一个 `QPainter`), 不同的是我们要使用 `PageData::paintPages()` 方法。在本节中, 将介绍那些用来绘制页面的 `PageData` 方法, 在本节的几个小节中, 将说明如何用 `paintPage()` 和 `paintPages()` 方法绘制 PDF、PostScript、SVG 和像素图片文件。对于打印预览, 我们已经了解到页面预览对话框调用了 `PageData::paintPages()` 方法。使用 `QPainter` 来生成输出时, 输出和打印是没有区别的。

使用 QPainter 就意味着把计算(尺寸与位置等)的负担留给了我们,而不是留给 QTextDocument,但我们得到的回报却是获得了对绘制内容和位置的完全控制。

```
void PageData::paintPages(QPrinter *printer, bool noUserInteraction)
{
    if (noUserInteraction)
        printer->setPageMargins(25, 25, 25, 25, QPrinter::Millimeter);
    QPainter painter(printer);
    for (int page = 0; page < pages.count(); ++page) {
        paintPage(&painter, page);
        if (page + 1 < pages.count())
            printer->newPage();
    }
}
```

这个方法与我们在前面介绍的 PageData::populateDocumentUsingHtml() 和 PageData::populateDocumentUsingQTextCursor() 方法有着相同的基本结构。

如果页面输出时没有用户交互存在,我们需要设定一些页面边距,否则就使用用户设定。然后创建一个 QPainter,绘制每个单独页面。这里没有使用反走样,因为我们认为绘制到打印机和绘制到一个矢量格式文件(如一个 SVG 文件)是一样的。除了最后一页,我们为其他每一个页面调用 QPrinter::newPage();这样做就排除了刚被打印过的页面,并且载入一个新的页面,准备打印。newPage() 方法返回一个表示成功与否的布尔值标志,尽管我们已经选择了要忽略它。

```
void PageData::paintPage(QPainter *painter, int page)
{
    const OnePage &thePage = pages.at(page);
    int y = paintTitle(painter, thePage.title);
    y = paintItems(painter, y, thePage);
    paintHtmlParagraph(painter, y, thePage.descriptionHtml);
    paintFooter(painter, tr("- %1 -").arg(page + 1));
}
```

这个方法与前面我们看到过的方法(例如,PageData::addPageToDocument())在结构上相似,对于页面的每一部分,把工作传递到帮助方法。但这里使用的帮助方法的一个重要不同之处在于,除了最后一个以外,其他的都返回一个 y 坐标。这个坐标是位于帮助方法已打印的页面的最下端——这样,接下来的方法就可以继续打印到页面的下端,而不重复打印前面已打印的部分(虽然 paintHtmlParagraph() 方法为段落落的行返回 y 坐标,但我们忽略它,因为 paintFooter() 方法基于视口和脚注的高度确定它的 y 坐标,而不考虑已经绘制的内容)。

如果用这个方法把页面绘制到一幅像素图片,创建绘制器的调用将会打开反走样。

```
int PageData::paintTitle(QPainter *painter, const QString &title)
{
    painter->setFont(QFont("Helvetica", 24, QFont::Bold));
    QRect rect(0, 0, painter->viewport().width(),
        painter->fontMetrics().height());
    painter->drawText(rect, title, QTextOption(Qt::AlignCenter));
    return qRound(painter->fontMetrics().lineSpacing() * 1.5);
}
```

该方法使用非常大的字号水平居中打印这个标题。矩形框的 y 坐标值为 0,这是顶端边距的基线,绘制器也使用这个值定位字体的顶端。换句话说,文本的底端在 y + painter->fontMetrics().height() 处。

一旦文本绘制完成,为下一个要绘制的对象返回一个 y 坐标,它带有一点垂直间隙。这是通过使用拥有 1.5 倍行高的 y 坐标计算而来的,其结果是创建一个半行高度的间隙。

```
int PageData::paintItems(QPainter *painter, int y,
    const OnePage &thePage)
{
    const int ItemHeight = painter->viewport().height() / 3;
```

```

const int ItemWidth = painter->viewport().width() / 2;
paintItem(painter, QRect(0, y, ItemWidth, ItemHeight),
          thePage_filenames.at(0), thePage_captions.at(0));
paintItem(painter, QRect(ItemWidth, y, ItemWidth, ItemHeight),
          thePage_filenames.at(1), thePage_captions.at(1));
y += ItemHeight;
paintItem(painter, QRect(0, y, ItemWidth, ItemHeight),
          thePage_filenames.at(2), thePage_captions.at(2));
paintItem(painter, QRect(ItemWidth, y, ItemWidth, ItemHeight),
          thePage_filenames.at(3), thePage_captions.at(3));
return y + ItemHeight + painter->fontMetrics().height();
}

```

在 `outputsampler` 程序中, 我们想要为“项”(图片及其伴随的标题)绘制一个 2×2 的表格。这个方法用来计算每个项将占用的矩形框, 并把实际的绘制内容传递到 `paintItem()` 帮助方法。

绘制器的视口已考虑了页面的边距, 因此可以用视口直接进行计算。我们已说过, 此处的每一项应该占用页面高度的三分之一以及宽度的一半。

在这段代码的最后返回一个 y 坐标, 它的值是两个项的高度加上 `QPainter::fontMetrics().height()`, `QPainter::fontMetrics().height()` 是它当前所拥有字体的字符高度。另一个可选的方法是使用当前字体的行高 (`QFontMetrics::lineSpacing()`)。

```

void PageData::paintItem(QPainter *painter, const QRect &rect,
                        const QString &filename, const QString &caption)
{
    painter->drawRect(rect);

    const int Margin = 20;
    painter->setFont(QFont("Helvetica", 18));
    const int LineHeight = painter->fontMetrics().lineSpacing();

    QRect imageRect(rect);
    imageRect.adjust(Margin, Margin, -Margin, -(Margin + LineHeight));
    QSvgRenderer svg(filename);
    QSize size(svg.defaultSize());
    size.scale(imageRect.size(), Qt::KeepAspectRatio);
    imageRect.setSize(size);
    const int Xoffset = (imageRect.width() - size.width()) / 2;
    imageRect.moveTo(imageRect.x() + Xoffset, imageRect.y());
    svg.render(painter, imageRect);

    int y = rect.y() + rect.height() - LineHeight;
    QRect captionRect(rect.x(), y, rect.width(), LineHeight);
    painter->drawText(captionRect, caption,
                     QTextOption(Qt::AlignCenter));
}

```

我们把每一项分成三部分绘制: 一个矩形轮廓、一幅图像和一个标题。矩形很容易绘制, 因为它是作为一个参数传递的。我们想让图像适合矩形的内部尺寸, 而且不与标题相冲突, 因此首先要设定一个边距以及标题的字体。一旦设定了字体, 就可以使用绘制器的字体矩阵来为文本计算行高了。

我们更倾向于使用 SVG 图像, 因为它们在缩放时不降低图像质量。首先基于项的矩形框为图片创建一个矩形框。然后为图像的矩形框尺寸减去边距尺寸, 高度除外 (实际上是它的底部 y 坐标值), 它的尺寸可以通过减去边距加上标题高度得到。

我们使用 `QSvgRenderer` 在绘制器上载入并绘制 SVG 图像。一旦图像被载入, 获取它的默认 (本来的) 尺寸, 然后缩放它以适应图像矩形框的内部尺寸, 在保持高宽比的情况下, 使用 `Qt::KeepAspectRatio` 来缩小尺寸以做到匹配。其他有效的枚举值 (这里是类比 `Qt::KeepAspectRatio`) 是 `Qt::IgnoreAspectRatio` 和 `Qt::KeepAspectRatioByExpanding`; 放大的那个枚举值 (就是指 `Qt::KeepAspectRatioByExpanding`) 使得图像放大到它的某个尺寸 (宽度或高度) 匹配已定义的尺寸, 另一个尺寸放大到超过给定限度时 (如果需要), 保持图像的高宽比。

既然我们想让图像水平居中,那么就要计算一个合适的 x 偏移。然后设定图像矩形框的尺寸为缩放后的尺寸并移动矩形框,以让它水平居中。接下来调用 `QSvgRenderer::render()` 方法,在使用图片矩形框的绘制器上绘制 SVG 图像。

如果使用的是像素图片而不是 SVG 图像,绘制图像的代码几乎是完全相同的。

```
QImage image(filename);
QSize size(image.size());
size.scale(imageRect.size(), Qt::KeepAspectRatio);
imageRect.setSize(size);
const int Xoffset = (imageRect.width() - size.width()) / 2;
imageRect.moveTo(imageRect.x() + Xoffset, imageRect.y());
painter->drawImage(imageRect, image);
```

唯一的区别在于,我们使用 `QImage` 而不是 `QSvgRenderer` 来载入图片,并获取它的尺寸,并且使用 `QPainter::drawImage()` 而不是 `QSvgRenderer::render()` 来绘制图像。我们可以考虑的一个改进是,如果图像的尺寸大于图像矩形框的尺寸,仅缩放像素图片的矩形框(包括已绘制的像素)(注意,在源代码中,我们使用了一个 `#define`,它用来在使用嵌入的 SVG 和嵌入的像素图片之间进行切换,默认情况下使用 SVG 图像)。

一旦绘制了图像,就要绘制标题。获取传入矩形框的 y 坐标值,将传入的“ y 坐标值加上矩形框底部的坐标值,然后减去(比如向上移动)一行的高度就可以得到必需的 y 坐标值。然后为标题创建一个合适的矩形框,在矩形框的中间(水平和垂直的)绘制标题。这样就可以保证标题被绘制在矩形框的底部,并且在标题的顶部和图片的底部之间留有一个间隙(边距的尺寸)。这意味着,标题的字母下部将到达矩形框(并且十分接近开始时绘制的矩形框的轮廓)的底部;我们把在文本下面添加一些边距留做练习。

当绘制 HTML 文本的段落时可以把 HTML 给予 `QTextDocument`。这样我们就有两个用于渲染文档内容的选择。较容易的办法是使用 `QTextDocument` 在绘制器上直接渲染 HTML;较难的办法是把 `QTextDocument` 作为一个富文本片段容器,我们自己遍历这些片段,绘制每一个片段。下面将演示这两种方法,它们的代码都是从创建 `QTextDocument` 开始的(源代码包括了这两种方法的实现,在编译时使用 `#define` 在它们之间进行转换)。

```
int PageData::paintHtmlParagraph(QPainter *painter, int y,
                                const QString &html)
{
    const QFont ParagraphFont("Times", 15);
    painter->setFont(ParagraphFont);
    QTextDocument document;
    document.setHtml(html);
```

首先设定一个默认的段落的字体;如前所述,它的字号比常用的大,目的是为了在屏幕截图中能显示得更清晰。在绘制器上设置字体,这样就可以使用绘制器的字体尺度了。然后创建 `QTextDocument`,并用传入的 HTML 段落填充它。

```
document.setDefaultFont(ParagraphFont);
document.setUseDesignMetrics(true);
document.setTextWidth(painter->viewport().width());
QRect rect(0, y, painter->viewport().width(),
           painter->viewport().height());
painter->save();
painter->setViewport(rect);
document.drawContents(painter);
painter->restore();
return y + document.documentLayout()->documentSize().height() +
        painter->fontMetrics().lineSpacing();
}
```

首先给予 `QTextDocument` 与绘制器相同的默认字体。通知文档以使用设计尺寸,因为这样会产生出高质量的效果。还要把文档的宽度限制到视口的宽度——这是良好布局的关键所在(但是,如果没有页面实际尺寸的限制,可以使用 `QTextDocument::idealWidth()`)。`QTextDocument::drawContents()` 方法可以接受剪贴下来的矩形框的第二个参数,但这里不需要它。我们想在页面靠下三分之二处绘制文本,但没有办法告诉 `QTextDocument` 去这样做。因此要保存绘制器的状态,改变它的视口,让视口成为一个矩形框,它从我们想要的 y 坐标开始,矩形框的宽度为页面的宽度。然后告诉文档在绘制器上绘制它的内容(这些将正好在我们想要的地方,也就是绘制器新的视口中发生),然后恢复绘制器的初始视口。

最后把文档高度(现在已经知道,因为要绘制内容,文档已经显示了它本身)加上一行的间距,这就是段落后的任何内容(如另一个段落)的 y 坐标。

使用 `QTextDocument` 来绘制一个 HTML 段落的第二种方法是让文本绘制自己。如果这样做,实质上是把 `QTextDocument` 当做一个 HTML 语法分析器使用,HTML 语法分析器把 HTML 转换为一个内部的文档结果,然后遍历并绘制每个单独的词。我们将介绍 `paintHtmlParagraph()` 方法第二部分的另一个版本,以此来说明这些是如何完成的。

```
QTextBlock block = document.begin();
Q_ASSERT(block.isValid());
int x = 0;
for (QTextBlock::iterator i = block.begin(); !i.atEnd(); ++i) {
    QTextFragment fragment = i.fragment();
    if (fragment.isValid()) {
        QTextCharFormat format = fragment.charFormat();
        foreach (QString word,
                 fragment.text().split(QRegExp("\\s+"))) {
            int width = painter->fontMetrics().width(word);
            if (x + width > painter->viewport().width()) {
                x = 0;
                y += painter->fontMetrics().lineSpacing();
            }
            else if (x != 0)
                word.prepend(" ");
            x += paintWord(painter, x, y, word, ParagraphFont,
                          format);
        }
    }
    y += painter->fontMetrics().lineSpacing();
}
```

段落存储在单独的 `QTextBlock` 中,其中的块包括一个或多个 `QTextFragment`,每一个 `QTextFragment` 都有自己的 `QTextCharFormat`。我们将遍历这些片段(如果段落的所有文本拥有相同的字符格式,则遍历单个片段),提取每个片段的格式,然后遍历它包含的所有词。用正则表达式分出“一个或多个空格”,这些表达式是用来把片段文本打断成为词的,这样可以把任何一个或多个空格序列当做一个空格对待。这样做具有一定意义,因为文本源为 HTML,它严格按照这种逻辑对待空格(顺便提一下,在循环内部创建正则表达式不像看上去那么耗资源;`QRegExp` 很小,足以记住最近使用的正则表达式,所以,在第一次构建正则表达式时,把它编译成内部的正则表达式格式,在后面的迭代中将使用已编译的格式)。

只要有了单词,我们就可以找到绘制器字体矩阵的宽度。如果单词不匹配当前行,重设 x 坐标为 0,为 y 坐标加上一行的高度;否则为单词预先考虑一个空格(以在水平方向上把它与本行的前一个单词分离开来),不改变坐标。然后调用我们自定义的 `paintWord()` 帮助方法,为 x 坐标加上该方法返回的偏移量。

最后返回 y 坐标加上一行的高度,这样就会在段落下面的行上绘制下一个对象。

```
int PageData::paintWord(QPainter *painter, int x, int y,
    const QString &word, const QFont &paragraphFont,
    const QTextCharFormat &format)
{
    QFont font(format.font());
    font.setFamily(paragraphFont.family());
    font.setPointSize(paragraphFont.pointSize());
    painter->setFont(font);
    painter->setPen(format.foreground().color());
    painter->drawText(x, y, word);
    return painter->fontMetrics().width(word);
}
```

对于我们绘制的每一个单词(一个单词或者是一个单词后面的一个空格),首先创建一个基于字体格式的字体。然后在绘制器上设定此字体,再设定绘制器的画笔为格式的前景色,这样就保留了已绘制的文本。然后在给定的 x 坐标和 y 坐标处绘制文本。最后返回绘制文本的宽度,这样,调用器就可以调整自身的 x 坐标方向的位置了。

```
void PageData::paintFooter(QPainter *painter, const QString &footer)
{
    painter->setFont(QFont("Helvetica", 11));
    painter->setPen(Qt::black);
    const int LineHeight = painter->fontMetrics().lineSpacing();
    int y = painter->viewport().height() - LineHeight;
    painter->drawLine(0, y, painter->viewport().width(), y);
    y += LineHeight / 10;
    painter->drawText(
        QRect(0, y, painter->viewport().width(), LineHeight),
        footer, QTextOption(Qt::AlignCenter));
}
```

为脚注设定不同的字体,把画笔的颜色设为默认色。绘制一个穿越页面的水平线,它的位置在下边距上面那一行。然后在线的下方居中位置绘制脚注(纯)文本。

现在,我们已完成了不使用 QTextDocument 而绘制一个文档的代码的说明。如果考虑到代码的行数,我们可能就会发现,对于同样的示例文档,使用带有原生 HTML 的 QTextDocument 产生的代码最少,绘制产生的代码最多。但这会因文档的不同而发生很大变化,因此我们推荐使用最适应实际情况的方法,而不是能产生最少代码的方法。

我们已经看到的方法都是从 PageData::paintPages() (它有一个 QPainter 参数)调用的,它(指 PageData::paintPages())依次调用 PageData::paintPage() (它有一个 QPainter 参数)。这意味着,就像我们将在接下来这个比较短的小节中看到的那样,这些方法可以用来在任何打印机或绘制设备上绘制示例文档。

10.4.1 绘制 PDF 或者是 PostScript

我们可以直接绘制一个 PDF 或是 PostScript 文档;所有需要的东西就是一个以 .pdf 或 .ps 结束的文件名。

```
bool MainWindow::paintPdfOrPs(const QString &filename)
{
    Q_ASSERT(filename.endsWith(".ps") || filename.endsWith(".pdf"));
    QPainter printer(QPrinter::HighResolution);
    printer.setOutputFileName(filename);
    printer.setOutputFormat(filename.endsWith(".pdf")
        ? QPrinter::PdfFormat : QPrinter::PostScriptFormat);
    pageData.paintPages(&printer);
    return true;
}
```

首先创建一个 `QPrinter`, 但不是使用打印机来打印纸质页面, 而是把它设定为“打印到文件”, 给予 `QPrinter` 一个要打印到的文件名以及一个要使用的格式(从结构上看, 这些代码与前面看到的 `exportPdfOrPs()` 方法的代码相同)。

10.4.2 绘制 SVG

我们可以用 `QSvgGenerator` 绘制 SVG 文件, 给予一个将要写 SVG 文件的文件名和一个页面尺寸就可以了(即便 SVG 图像是可缩放的, 但通常的做法是为它们提供一个默认或“自然”尺寸)。对于 SVG 文件, 我们已选择为每个单独页面而不是整个多页文档创建一个单独的 SVG 图像。

```
bool MainWindow::paintSvg(const QString &filename)
{
    Q_ASSERT(filename.endsWith(".svg"));
    QSvgGenerator svg;
    svg.setFileName(filename);
    QRect rect = printer.pageRect().adjusted(25, 25, -25, 25);
    svg.setSize(rect.size());
    QPainter painter(&svg);
    painter.setViewport(rect);
    pageData.paintPage(&painter, 0);
    return true;
}
```

绘制过程很简单: 创建一个 `QPainter`, 绘制一个页面(页面 0)。此处我们选择偏移这个图像以提供一些边距(从结构上看, 这些代码与前面看到的 `exportSvg()` 方法的代码相同)。

10.4.3 绘制像素图片

`QPainter` 可以绘制能被渲染为像素图片或者是矢量图的图像。因此, 一个 `QPainter` 可以绘制到 `QImage` 而 `QImage` 可以保存在 Qt 支持的任何像素图片格式中(这个格式列表由静态 `QImageWriter::supportedImageFormats()` 方法返回)。

```
bool MainWindow::paintImage(const QString &filename)
{
    QImage image(printer.paperRect().size(), QImage::Format_ARGB32);
    QPainter painter(&image);
    painter.setRenderHints(QPainter::Antialiasing |
                           QPainter::TextAntialiasing);
    painter.fillRect(painter.viewport(), Qt::white);
    painter.setViewport(printer.pageRect());
    pageData.paintPage(&painter, 0);
    return image.save(filename);
}
```

至此, 我们创建所需要尺寸的 `QImage`、建立反走样、为背景填充白色以清除它——可以容易地使用 `Qt::transparent`——然后绘制一个页面(页面 0)。就像绘制 SVG 图像一样, 我们选择为一个单独页面创建一幅单独的图像(从结构上看, 这些代码与前面看到的 `exportImage()` 方法的代码相同)。

至此, 我们已经说明了如何打印和输出包括富文本、图像和复杂格式(如表格)的文档。还说明了如何打印和输出为 SVG 图像的 `QTextDocument` 的单个页面以及其他 Qt 支持的标准像素图片格式, 还说明了如何直接只用 `QPainter` 以达到同样的目的。同样也说明了如何打印和输出整个多页面 `QTextDocument` 文档到 PDF、PostScript、Open Document Format 和 HTML。还有如何直接用 `QPainter` 绘制文档以产生 PDF 和 PostScript。

在本章中, 我们广泛地使用了 `QPainter` 来绘制文档, 从上一章可知, `QPainter` 还可以用做绘制自定义窗口部件和图形。但 Qt 为复杂图形提供了相对 `QPainter` 而言更为强大的可选方法: 图形/视图架构, 它是后面两章的主题。

第 11 章 创建图形/视图窗口

- 图形/视图架构
- 图形/视图窗口部件和布局
- 图形项简介

通过自定义 `QWidget` 派生类,实现 `paintEvent()` 并使用 `QPainter`,我们可以随心所欲地绘制任何需要的内容。这种方法对于自定义的窗口部件(widget)很理想,但如果要绘制大量独立的项(item)就不怎么方便了,尤其是当想要向用户提供与图形项交互能力时。比如曾有用户创建使用成千上万的自定义窗口部件作为图形项的图形应用,虽然窗口部件的绘制非常快,但这种状况下处理单个鼠标点击轻易就耗费掉了几乎所有的 CPU 处理能力。幸好 Qt 4.2 引入了图形/视图架构,完美地满足了基于项的高性能绘制和交互的需求。

Qt 4 的图形/视图架构原本的设计是取代并超越 Qt 3 的 `QCanvas` 类,但它已经远远超过了画布(canvas)的功能。实际应用中有些应用程序用 `QGraphicsView` 作为主窗口的中心窗体,把所有提供用户界面的窗口部件作为图形项放在视图中。

在 11.1 节我们先从图形/视图架构的概览开始,涵盖一些 Qt 4.6 中新加入的关键变化。在 11.2 节将查看一个用 `QGraphicsView` 作为主窗口的中心窗体的应用程序,`QGraphicsView` 中既包含窗口部件项又包含传统的图形项。最后在 11.3 节我们将介绍一个简单的 `QGraphicsItem` 的子类和 `QGraphicsItem` 的 API。

在下一章将介绍一个更传统的图形/视图应用程序(一个基本的绘图程序)更深入地了解大部分图形/视图类,还会介绍更多关于如何创建自定义图形项的例子。顺便提一句,这一章里出现的例子将在第 13 章中使用一些 Qt 4.6 中才有的特性创建一个修改版。

11.1 图形/视图架构

与 Qt 的模型/视图(model/view)架构十分相似,图形/视图架构有一个作为模型的不可见类(`QGraphicsScene`)容纳项的数据,同时用另一个类(`QGraphicsView`)来可视化显示数据。如有需要,我们可以在多个不同的视图内可视化显示同一场景。图形场景包含了从抽象类 `QGraphicsItem` 派生而来的项。

从最初被引入到 Qt 中至今,得益于大量的开发努力,图形/视图架构在速度和功能上都有了很大进步。场景可缩放、旋转并可打印,渲染则可以使用 Qt 的渲染引擎或者使用 OpenGL。该架构还支持动画和拖放操作。图形场景可以用来呈现任何内容,从几个项直至成千上万乃至更多的项。

Qt 提供了可以直接使用的众多预定义图形项类型,如图 11.1 所示。多数类的名称可顾名思义,但我们会提及一些名称不那么显而易见的类。`QGraphicsPathItem` 代表一个 `QPainterPath`,本质上是一个任意的形状,由所有基本的 Qt 可绘制的物体,包括弧线、贝济埃曲线、弦、椭圆形、线段、矩形和文本——组合而成。`QGraphicsSimpleTextItem` 代表一个纯文本(plain text)片段,`QGraphicsTextItem` 则代表 Qt 的富文本(rich text)片段(可以用 HTML 语言描述;我们在前两章中对富文本进行了讨论)。`QGraphicsWidget` 类是创建在图形场景中存在的自定义窗口部件的基类。通过把窗口部件添加到 `QGraphicsProxyWidget`,再把代理窗口部件(proxy widget)加入到场景中,我们也能在场景中嵌入标准的从 `QWidget` 派生的窗口部件。使用代理窗口部件(或者直接使用窗口部件)比较慢,但至于会不会

慢到被注意到的程度还要看应用程序^①。Qt 4.6 引入了 `QGraphicsWebView` 类,提供 `QWebView` 类(我们在第 1 章中讨论了这个类)的图形项版本,用于在场景中呈现网络内容(Web content)。

对于项不多的场景,可以使用在 Qt 4.6 中引入的 `QGraphicsObjects` 类,对于 Qt 4.5 及更早的版本,可以把 `QObject` 和 `QGraphicsItem` 两个类作为基础派生自定义的项。这会增加每个项的开销(如项会占用更多的内存),但可以提供对信号和槽以及 Qt 的属性系统的支持。对于有很多很多项的场景来说,通常最好是用轻量级的 `QGraphicsItem` 类作为数目众多的自定义项的基础,仅对数目较少的项使用 `QGraphicsObject`。

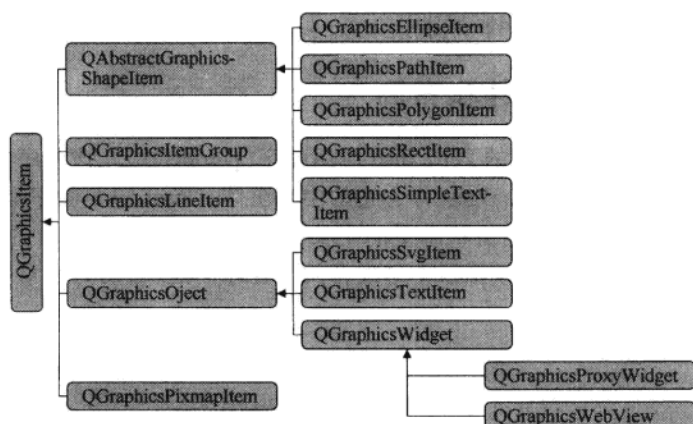


图 11.1 Qt 的 `QGraphicsItem` 类层次结构

虽然图形视图类的每个项都有一个 z 值, z 值较大的项会被绘制在 z 值较小的项之上,但图形视图类本质上是二维的。冲突检测基于项的 (x, y) 坐标位置。除了冲突信息之外,场景还可以告诉我们哪些项包含一个特定的点,哪些项在某特定区域中以及哪些项被选中。场景还有一个有用的前景层,例如在场景中绘制一个覆盖所有项的网格;场景还有一个背景层在所有项之下绘制,可用于提供背景图或者背景色。

项可以是场景或者另一个项的子对象,和 Qt 窗口部件的父子关系十分相似。当一个项应用了变换(transformation),该项的所有子对象自动地应用该变换,递归应用至最深层次的子孙对象。这意味着如果一个项被移动(如被用户拖动),它的所有子项(及子项的子孙)将和它同时被拖动。通过调用 `QGraphicsItem::setFlag(QGraphicsItem::ItemIgnoresTransformations)` 可以让子项忽略父项的变换。除此以外更常用的标志包括设置项可移动、选中和获得焦点的标志(所有的标志在表 11.6 和表 11.7 中列出)。通过把项设置为 `QGraphicsItemGroup` 的子项,可以把它们分组;这对于创建机动的项集合很有用。

图形视图类使用三种不同的坐标系统,虽然在实际的应用中通常只关心其中的两种。视图使用物理坐标系统,场景使用依照在构造函数中传入的 `QRectF` 定义的逻辑坐标系统。Qt 自动进行从场景坐标到视图坐标的映射。重点在于场景用窗口(window)的(逻辑)坐标而视图用视口(viewport)的(物理)坐标。所以我们放置项时用的是场景的坐标系统。第三个坐标系统是项使用的坐标系统,该系统非常实用,因为它是中心在 $(0, 0)$ 点的逻辑坐标系统。每个项的 $(0, 0)$ 点实际上在该项场景中所在位置的(除了文本项,它的原点在项的左上角)。这表示在实际使用中我

^① 关于在场景中使用窗体部件和代理的性能问题的讨论,请参阅文章 labs.qt.nokia.com/blogs/2010/01/11/qt-graphics-and-performance-the-cost-of-convenience。

们总是可以以它的中心点来绘制项,不需要在意父项应用到当前项的变换,因为场景会自动处理。另外需要注意的是,在 Qt 中, y 坐标向下增长,例如点(5,8)在点(5,14)的上方 6 像素位置。图 11.2 演示了场景和项坐标之间的关系。

图形/视图架构的某些方面的行为在 Qt 4.5 和 Qt 4.6 之间发生了一些变动,这些变动总结在“Qt 4.6 图形/视图行为的变化”的阴影部分中。

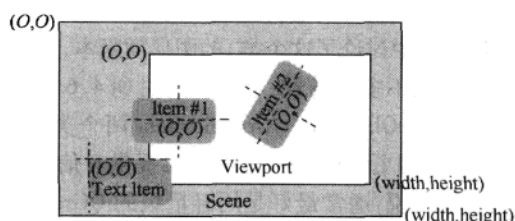


图 11.2 图形项使用本地逻辑坐标

11.2 图形/视图窗口部件和布局

本节我们将查看图 11.3 展示的 Petri Dish 应用程序 (petridish1)。该程序的 MainWindow 类从 QMainWindow 派生,用 QGraphicsView 作为它的中心窗体。Petri Dish 程序是一个对话框风格的模拟器程序,模拟“细胞”在非拥挤状态下的生长,在分离、拥挤和过大时的收缩和小细胞随机“死亡”。我们不过多讨论关于模拟器本身或者关于程序的逻辑,因为本章的重点是 Qt 的图形/视图架构。

Qt 4.6 图形/视图行为的变化

图形/视图类在 Qt 4.5 和 Qt 4.6 之间经历了相当大的发展,性能显著提高。这些底层变化带来的结果之一是为了达成最大可能性的优化,不可避免地改变了某些用户可见的行为。关键的行为改变如下:

- QStyleOptionGraphicsItem 的公有变量 (QRectF 类型的 exposedRect) 保存项在项坐标系统的曝光矩形区域 (exposed rectangle), 但仅有设置了 ItemUsersExtendedStyleOption 标志的图形项才会设置这个变量。
- QStyleOptionGraphicsItem 的 levelOfDetail 和 matrix 变量都已经废弃不用。在 Qt 4.6 中正确地获得详细信息级别的方法是用静态函数 QStyleOptionGraphicsItem::levelOfDetailFromTransform()。
- QGraphicsView 不再调用 QGraphicsView::drawItems() 或者 QGraphicsView::drawItem(), 除非你设置 QGraphicsView::IndirectPainting 这个“优化”标志 (不推荐使用)。
- QGraphicsItem 不再为位置和变换的变化调用 itemChange(), 如果要接受变化的通知, 设置 QGraphicsItem::ItemSendsGeometryChanges 标志 (QGraphicsWidget 和 QGraphicsProxyWidget 默认已设置了这个标志)。

即使设置了 ItemSendsGeometryChanges 标志, 当变换发生时, 仅当使用了 setTransform() 函数时, itemChanged() 才会被调到。从 Qt 4.7 开始当调用 setRotation()、setScale() 或 setTransformOriginPoint() (这些函数在 Qt 4.6 引入) 时, 如果设置了这个标志, itemChange() 函数也会被调用。

这些变化会影响某个程序到何种程度 (甚或是否会影响该程序) 取决于程序用到了哪些图形/视图的特性。以本书提供的例子来说, 下一章的页面设计器 (Page Designer) 程序就受到了上述最后一项的影响。

在这里将查看主窗口相关的成员函数(或者从这些函数中提取出来的摘要)来展示如何创建一个基于主窗口的图形场景。下一节将看看“细胞”项(从 `QGraphicsItem` 派生),重点放在创建自定义图形项的基本步骤以及 `QGraphicsItem` 的 API,忽略无关的模拟逻辑(源码在 `petridish1` 子目录中)。

程序有“开始”、“暂停/继续”、“停止”和“退出”这些控制模拟的按钮,用户可以设置初始的细胞数和是否显示细胞的 ID——对于细胞小到看不到的情况比较有用(如图 11.3 所示,初始细胞数在模拟过程中为禁用状态)。

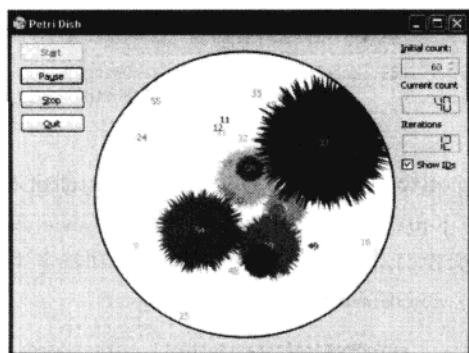


图 11.3 Petri Dish 应用程序

界面使用了两个 `QLCDNumber` 显示剩余多少细胞和模拟进行了多少次。

我们从主窗口的构造函数看起,然后继续看几个所关心的与图形/视图编程相关的帮助函数。

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), simulationState(Stopped), iterations(0)
{
    scene = new QGraphicsScene(this);
    scene->setItemIndexMethod(QGraphicsScene::NoIndex);

    createWidgets();
    createProxyWidgets();
    createLayout();
    createCentralWidget();
    createConnections();

    startButton->setFocus();
    setWindowTitle(QApplication::applicationName());
}
```

`QGraphicsScene` 的创建看上去有点不同寻常,因为这里没有指定场景的尺寸范围。虽然知道所需的高度(高度足以容纳培养皿加上边缘的空白),但是宽度要取决于窗口部件的宽度,所以等到窗口部件创建并布局完毕再设置尺寸。

当项被添加到场景、移动或者从场景中移除,就需要重新计算位置。例如,当项被添加到场景中的可见部分,就必须将它重绘;当可见的项被移动或移除,不管是被它遮挡的还是现在显露出来的项都必须被重绘。对于那些包含众多静态项的场景来说,通过使用 `QGraphicsScene::BspTreeIndex` (Binary Space Partitioning 二分空间分区)这种索引方法,可以相当程度地加快这些计算;但对于包含众多被添加、移动、移除的项的动态场景来说,最好是关掉索引(如这里所做),因为使用索引带来的耗费超过了它节省的开销。

整本书的编码风格的共同之处在于,在构造函数中调用帮助函数执行窗口部件大部分的初始化。我们在主窗口的中心窗体中使用一个图形场景,所有的帮助函数是相关的,所以把它们放在一起展示和讨论(但尽可能省略重复的代码)。

```
void MainWindow::createWidgets()
{
    startButton = new QPushButton(tr("Start"));
    pauseOrResumeButton = new QPushButton(tr("Pa&use"));
    pauseOrResumeButton->setEnabled(false);
    stopButton = new QPushButton(tr("Stop"));
    quitButton = new QPushButton(tr("Quit"));

    QString styleSheet("background-color: bisque;");
```

```

initialCountLabel = new QLabel(tr("Initial count:"));
initialCountLabel->setStyleSheet(styleSheet);
...
AQP::accelerateWidgets(QList<QWidget*>() << startButton
    << stopButton << quitButton << initialCountLabel
    << showIdsCheckBox);
}

```

这个程序使用标准的 `QWidget`, 控件的创建不存在任何特别之处。仅有一处略有不同, 即为窗口部件(不包括按钮)提供了一个样式表(style sheet)以提供统一的背景色。按钮没有设置样式表, 是因为我们希望它们能保留特定平台和特定主题的外观。

```

void MainWindow::createProxyWidgets()
{
    proxyForName["startButton"] = scene->addWidget(startButton);
    proxyForName["pauseOrResumeButton"] = scene->addWidget(
        pauseOrResumeButton);
    ...
}

```

因为场景的视图是主窗口的中心窗体, 所以所有的窗口部件都要添加到场景中。可以很容易地采取另一种解决方法——例如, 使用一个普通窗体作为中心窗体, 赋予其一个 `QHBoxLayout`, 在其中用一个 `QVBoxLayout` 容纳按钮和 `QGraphicsView`, 用另一个 `QVBoxLayout` 容纳其他窗口部件。但为了展示这种方法的可行性, 我们选定 `QGraphicsView` 本身作为中心窗体, 在其中同时放置图形项和其他窗口部件。

向场景中添加标准 `QWidget` 的方法是为每个 `QWidget` 创建一个 `QGraphicsProxyWidget`, 然后把代理(proxy)加到场景中。我们使用的成员函数是 `QGraphicsScene::addWidget()`, 它创建一个 `QGraphicsProxyWidget`, 代表以参数方式传入的窗口部件, 并返回一个代理窗体的指针作为结果。为方便起见, 保存一个以窗口部件名称为键值的哈希表(hash), 值为代理窗体的指针, 把所有创建的代理窗体添加到哈希表中(哈希表在头文件声明为 `QHash<QString, QGraphicsProxyWidget*> proxyForName;`)。

一旦这些部件和它们的代理都创建完毕, 我们就可以对它们进行布局了。如果用 Qt 的标准布局, 做法类似, 除了必须用图形-场景对应的布局类。我们将查看 `createLayout()` 函数的两个部分, 一是看布局的创建, 二是看场景规格尺寸的设置。

```

const int DishSize = 350;
const int Margin = 20;

void MainWindow::createLayout()
{
    QGraphicsLinearLayout *leftLayout = new QGraphicsLinearLayout(
        Qt::Vertical);
    leftLayout->addItem(proxyForName["startButton"]);
    leftLayout->addItem(proxyForName["pauseOrResumeButton"]);
    leftLayout->addItem(proxyForName["stopButton"]);
    leftLayout->addItem(proxyForName["quitButton"]);

    QGraphicsLinearLayout *rightLayout = new QGraphicsLinearLayout(
        Qt::Vertical);
    foreach (const QString &name, QStringList()
        << "initialCountLabel" << "initialCountSpinBox"
        << "currentCountLabel" << "currentCountLCD"
        << "iterationsLabel" << "iterationsLCD"
        << "showIdsCheckBox")
        rightLayout->addItem(proxyForName[name]);

    QGraphicsLinearLayout *layout = new QGraphicsLinearLayout;
    layout->addItem(leftLayout);
    layout->setItemSpacing(0, DishSize + Margin);
}

```



```
layout->addItem(rightLayout);
QGraphicsWidget *widget = new QGraphicsWidget;
widget->setLayout(layout);
scene->addItem(widget);
```

`QGraphicsLinearLayout` 类是一个图形/视图的布局类, 对应于 `QHBoxLayout` 和 `QVBoxLayout` 继承的基类 `QBoxLayout`。我们用 `QBoxLayout::addWidget()` 添加窗口部件, 而用 `QGraphicsLinearLayout::addItem()` 添加项, 除此以外, 两者的 API 非常接近。 `addItem()` 函数添加一个 `QGraphicsLayoutItem` (这个类是 `QGraphicsWidget`、进而是 `QGraphicsProxyWidget` 的基类) 到布局。还有一个和 `QGridLayout` 对应的 `QGraphicsGridLayout` 类。Qt 4.6 引入 `QGraphicsAnchorLayout` 类, 实现了 Qt 中从未出现过的全新的布局方法——基于窗口部件互相之间的相对位置、边缘和布局占用矩形的各个角来放置控件。

这个函数中我们创建了三个 `QGraphicsLinearLayout`, 第一个布局用于在左边纵向排布一系列按钮的代理控件, 第二个布局用于在右边垂直排布一系列代理窗口部件, 第三个布局用于提供一个从左端到右端的水平布局包含左边的布局、占位符 (提供放置培养皿的空间) 和右边的布局。布局的情形如图 11.4 所示。

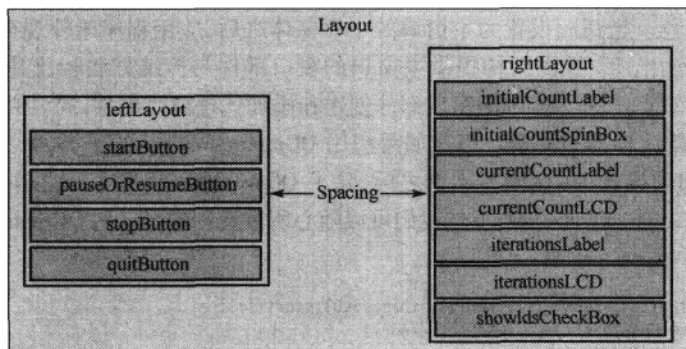


图 11.4 Petri Dish 的主窗口布局

完成创建布局之后紧接着创建一个新的“空白” `QGraphicsWidget`。这个类本身没有可见的表现, 它特别设计为自定义图形/视图窗口部件的基类, 或者是它放在这里的目的 (为了把一个或多个子窗体组织在布局中)。创建完这个窗体, 在其中设置全局布局, 然后把这个窗体添加到场景中。结果是所有的布局和代理窗口部件都被重定父级 (reparent)——比如代理窗口部件成为场景的子窗体 (在调用 `QGraphicsScene::addWidget()` 时, 窗口部件重定父级到它们的代理)。

```
int width = qRound(layout->preferredWidth());
int height = DishSize + (2 * Margin);
setMinimumSize(width, height);
scene->setSceneRect(0, 0, width, height);
}
```

我们把场景宽度设置到足够显示布局的合适宽度, 高度设置到足够显示培养皿和垂直空白。还设置了主窗口的最小尺寸, 这样它不会缩小得过小而无法适当地显示培养皿和窗口部件。

```
void MainWindow::createCentralWidget()
{
    dishItem = new QGraphicsEllipseItem;
    dishItem->setFlags(QGraphicsItem::ItemClipsChildrenToShape);
    dishItem->setPen(QPen(QColor("brown"), 2.5));
    dishItem->setBrush(Qt::white);
}
```

```

dishItem->setRect(pauseOrResumeButton->width() + Margin,
                Margin, DishSize, DishSize);

scene->addItem(dishItem);

view = new QGraphicsView(scene);
view->setRenderHints(QPainter::Antialiasing|
                    QPainter::TextAntialiasing);
view->setBackgroundBrush(QColor("bisque"));
setCentralWidget(view);
}

```

场景创建完成并填充了窗口部件(或者填充窗口部件代理)之后,上面这个函数被调用以创建培养皿和视图,这样就完成了应用程序外观的设置。

我们从创建一个新的椭圆图形项开始——虽然在此例中它实际上是一个圆形,因为我们把宽度和高度设置成一样。把这个项设置为剪切(clip)它的子项,所有的模拟细胞都创建为培养皿的子项,这样可以保证培养皿区域之外的细胞不会显示,跨在培养皿边缘的细胞仅有边缘以内的部分被绘制。我们设置培养皿的矩形 x 坐标为左边布局中按钮的宽度加上一些空白, y 坐标在上方保留少量空白区域。培养皿项创建完毕后就把它加到场景中。

创建一个打开反走样(antialiasing)功能标准的 QGraphicsView,并使用为一些窗口部件的样式表设置相同的背景色。把视图设置为主窗口的中心窗体之后,应用程序的外观全部完成。

从结构上看,使用图形/视图架构提供主窗口的窗口部件与传统方法相比并无特别不同之处。唯一比较大的区别在于,必须为实际的窗口部件创建和添加代理窗口部件,同时必须使用图形/视图特定的布局而不能使用标准布局类。当然如果想用 QGraphicsWidget 的子类,就不需要再创建代理了,因为可以直接加到场景中(在撰写本书之际,除了 QGraphicsProxyWidget 之外唯一的 QGraphicsWidget 子类只有 QGraphicsWebView,虽然我们可以随心所欲地创建自己的 QGraphicsWidget 子类)。

```

void MainWindow::createConnections()
{
    connect(startButton, SIGNAL(clicked()), this, SLOT(start()));
    connect(pauseOrResumeButton, SIGNAL(clicked()),
            this, SLOT(pauseOrResume()));
    connect(stopButton, SIGNAL(clicked()), this, SLOT(stop()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(showIdsCheckBox, SIGNAL(toggled(bool)),
            this, SLOT(showIds(bool)));
}

```

这个函数与前面几章我们看见过多次的函数类似,建立了从(实际的)窗口部件的 clicked() 信号到对应的槽连接。这和图形/视图编程没有多大关系,但在这里列出是为了提供与第 13 章出现的 Qt 4.6 版本的例子的对照,在那个例子中使用了 QStateMachine 控制程序的行为,这样减少了槽的数量,逻辑也更简单。

11.3 图形项简介

QGraphicsItem 类是所有图形项的基类,虽然这个类提供了数量惊人的成员函数(Qt 4.6 中有超过 200 个),它仍有两个纯虚函数,所以不能被实例化,这两个函数是: boundingRect() 和 paint()。paint() 函数对应于 QWidget::paintEvent(), 必须被重新实现以绘制项。boundingRect() 函数在图形/视图架构中为项提供一个边界矩形——用于冲突检测和保证项仅在 QGraphicsView 的视口(viewport)中可见时才重绘。

如果要创建非矩形形状的自定义图形项,最好是同时重新实现 shape() 方法。这个方法返回一个精确描述项的轮廓的 QPainterPath,对于准确检测冲突和鼠标点击非常有用。

有很多可供重新实现的虚函数,包括 `advance()`、`boundingRect()`、`collidesWithItem()`、`collidesWithPath()`、`contains()`、`isObscuredBy()`、`opaqueArea()`、`paint()`、`shape()` 和 `type()`。所有的保护成员函数(除了 `prepareGeometryChange()`)也是虚函数,因此所有的图形项事件处理函数(event handlers)(包括 `contextMenuEvent()`、`keyPressEvent` 和鼠标事件)都可以被重新实现。所有这些函数都在表 11.1 至表 11.4 中进行了简要的说明。

如果需要一个自定义的形状,最简单的莫过于使用标准的 `QGraphicsItem` 子类之一,比如 `QGraphicsPathItem` 或者 `QGraphicsPolygonItem`。进而如果我们还想要这个形状有定制的行为,可以派生这个项,然后重新实现几个保护事件处理函数,如 `keyPressEvent()` 和 `mousePressEvent()`。如果只是想要自己绘制,我们可以直接从 `QGraphicsItem` 派生,重新实现 `boundingRect()`、`paint()` 和 `shape()` 方法,加上提供我们想要的行为所必须的事件处理函数。对于所有的 `QGraphicsItem` 子类来说,最好是提供一个 `Type` 枚举类型,并且重新实现 `type()` 函数,我们将在下面讨论。

下面简略浏览一下培养皿程序的“细胞”类单纯的与图形/视图相关的方面,这个类是从 `QGraphicsItem` 直接派生的。我们从头文件的定义开始,但省略私有部分。

```
class Cell : public QGraphicsItem
{
public:
    enum {Type = UserType + 1};
    explicit Cell(int id, QGraphicsItem *parent=0);

    QRectF boundingRect() const { return m_path.boundingRect(); }
    QPainterPath shape() const { return m_path; }
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option, QWidget *widget);
    int type() const { return Type; }
    ...
};
```

虽然重新实现 `type()` 方法或者提供一个 `Type` 枚举型并不是必需的,推荐为每个自定义的图像项子类提供这两者。这样可以很容易唯一地标志每个自定义图形项的类型,同时也意味着它们可以和 `qgraphicsitem_cast <>()` 转换 `QGraphicsItem` 指针为正确的 `QGraphicsItem` 子类的函数一起工作(`qgraphicsitem_cast <>()` 函数只支持从 `QGraphicsItem` 指针到子类的类型转换,不能从子类反向转换为 `QGraphicsItem` 指针。反向转换要用其他方法。在下一章将讨论图形项的类型转换)。

在这个例子中有个私有成员变量——`QPainterPath` 类型的 `m_path`(在模拟过程中动态改变形状)。有了这个路径,就能用它来提供项的边界矩形和项的形状。注意,虽然从绘制路径(`painter path`)计算一个边界矩形对于培养皿程序来说还算够快,但一般情况下不会太快。其他使用画刷路径的程序应该可以得益于缓存边界矩形的路径。

`shape()` 方法实现与否无关紧要,因为稍后将看到路径绘制仅用画刷(`brush`)而没用到画笔(`pen`)。如果要用画笔绘制路径的话——比如用椭圆路径和很粗的画笔做出的环形形状,那么得到的形状不够精确,因为轮廓线没有计入线条的粗细。这样的话用户点击轮廓线时就没有效果,因为它在椭圆“之外”。在这种情况下可以创建 `QPainterPathStroker` 并用画笔相关的方法(`setWidth()`、`setJoinStyle()`等)设置,然后调用 `QPainterPathStroker::createStroke()`,把绘制路径(`painter path`)作为参数传入。`createStroke()` 函数的返回值是原路径的轮廓线加上我们设置的画笔属性生成的一个新的绘制路径。

在构造函数(这里未列出)中对“细胞”设置了画刷和初始大小,然后调用一个私有方法(在此

未列出)创建初始形状。这样 `paint()` 方法就相对简单很多,因为它只需要绘制出路径即可(和可选的项标志 ID)。

```
void Cell::paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option, QWidget*)
{
    painter->setPen(Qt::NoPen);
    painter->setBrush(m_brush);
    painter->drawPath(m_path);
    if (s_showIds) {
        QPointF center = m_path.boundingRect().center();
        QString id = QString::number(m_id);
        center.setX(center.x() - (option->fontMetrics.width(id) / 2));
        center.setY(center.y() + (option->fontMetrics.height() / 4));
        painter->setPen(QPen());
        painter->drawText(center, id);
    }
}
```

首先设置绘制使用的画笔(`Qt::NoPen` 表示不绘制轮廓线),然后是画刷,最后绘制细胞的路径(“细胞”类还有一个布尔类型的静态变量——`s_showIds`,以及几个静态访问函数和一个 ID 成员变量——整型的 `m_id`,这些在这里均未列出)。如果项的 ID 需要显示出来,我们找到路径的中心,然后使用 `QPen()` 把 ID 绘制在水平中心、垂直顶端下方四分之一处。`QPen` 的默认构造函数创建一个像素粗的黑色实线装饰画笔(cosmetic pen)。当画笔忽略变换(transformation)时就被称为装饰画笔。

`QStyleOptionGraphicsItem *` 参数保存诸如项的曝光矩形区域(exposed rectangle)、字体的度量信息(font metrics)(在这里使用到的)和调色板之类的属性。`QWidget *` 参数很少使用到。

`paint()` 函数的实现对于 Petri Dish 程序来说已经够快了,但远远未到最优化。在本例中对路径的边界矩形做缓冲可能不太值得,因为细胞在模拟的每个循环中都会放大或者缩小。但细胞的标志从不改变,所以我们保存一个私有成员变量 `QString` 类型的 `m_idString`,在构造函数里创建,这样避免在 `paint()` 函数中调用 `QString::number()` 方法(该方法每次被调用都会额外分配内存),借此用内存的一点点额外消耗换取了速度的提升。计算字体尺度的宽和高也很慢,可以在构造函数中计算这个值并将它缓存起来。当然最好是对每个修改进行性能测试,以确保达到我们希望的成效。

细胞类没有重新实现任何一个事件处理函数,也没有设置任何如 `ItemIsMovable` 或 `ItemIsSelectable` 之类的标志,所以用户不能直接和细胞项进行交互。在后面的章节中将看到设置标志和实现事件处理函数的例子。本节的最后是总结 `QGraphicsItem` API 的表格。表 11.1 至表 11.4 总结了 `QGraphicsItem` 类的成员方法,表 11.5 至表 11.7 总结了成员方法中用到的重要的枚举值。

表 11.1 QGraphicsItem API(部分经过挑选的方法) #1

方 法	说 明
<code>advance()</code>	重新实现以实现动画;其他方法也可使用(例子参见第 12 章和第 13 章)
<code>boundingRect()</code>	重新实现以返回项坐标系表示的项的边界矩形;参见 <code>sceneBoundingRect()</code> 和 <code>shape()</code>
<code>childItems()</code>	返回当前项的直接子项列表(<code>Qt 4.4</code>)
<code>collidesWithItem(QGraphicsItem *, Qt::ItemSelectionMode)</code>	根据模式判断,当这个项与给定项冲突时返回 <code>true</code> ;参见 <code>Qt::ItemSelectionMode</code> 枚举值
<code>collidesWithPath(QPainterPath, Qt::ItemSelectionMode)</code>	根据模式判断,当这个项与给定路径冲突时返回 <code>true</code>

(续表)

方 法	说 明
<code>collidingItems(Qt::ItemSelectionMode)</code>	根据模式判断,返回与当前项冲突的所有项目列表
<code>contains(QPointF)</code>	如果点在项目范围之内则返回 <code>true</code>
<code>ensureVisible()</code>	如果有必要,强制 <code>QGraphicsView</code> 关联的、包含某个项的场景滚动以显示出该项
<code>group()</code>	返回该项所属的 <code>QGraphicsItemGroup</code> ,如果该项不属于任何组则返回 0
<code>hide()</code>	隐藏该项;参见 <code>show()</code> 和 <code>setVisible()</code>
<code>isObscuredBy(QGraphicsItem *)</code>	如果该项的边界矩形完全被给定的非透明项的形状所遮盖则返回 <code>true</code>
<code>isSelected()</code>	该项被选中时返回 <code>true</code> ;参见 <code>setSelected()</code>
<code>isVisible()</code>	如该项逻辑上可见则返回 <code>true</code> [(即使它被完全遮盖或者在视图 (view) 的视口 (viewport) 之外)]
<code>keyPressEvent(QKeyEvent *)</code>	重新实现这个函数以处理项的键盘按下事件,仅在设置了 <code>ItemsIsFocusable</code> 标志的情况下才会被调用
<code>mouseDoubleClickEvent(QGraphicsSceneMouseEvent *)</code>	重新实现这个函数以处理鼠标双击事件

表 11.2 QGraphicsItem API(部分经过挑选的方法) #2

方 法	说 明
<code>mouseMoveEvent(QGraphicsSceneMouseEvent *)</code>	重新实现这个函数以处理鼠标移动事件
<code>mousePressEvent(QGraphicsSceneMouseEvent *)</code>	重新实现这个函数以处理鼠标按下事件
<code>moveBy(qreal, qreal)</code>	按照给定参数水平和垂直方向移动该项
<code>opaqueArea()</code>	重新实现这个函数以返回显示该项不透明区域的绘制路径,在 <code>isObscuredBy()</code> 函数中使用
<code>paint(QPainter *, QStyleOptionGraphicsItem *, QWidget *)</code>	重新实现这个函数以绘制该项;参见 <code>boundingRect()</code> 和 <code>shape()</code> 函数
<code>parentItem()</code>	返回该项的父项或者 0
<code>pos()</code>	返回该项在它的父项坐标系中的位置——或当它无父项时返回场景坐标系的位置;参见 <code>scenePos()</code> 函数
<code>prepareGeometryChange()</code>	这个方法必须在改变项的边界矩形之前调用;它自动调用 <code>update()</code> 函数
<code>resetTransform()</code>	重置该项的变换矩形为恒等矩形 (identity matrix), 等于去除所有旋转、缩放和剪切
<code>rotation()</code>	返回该项的旋转度数 (-360.0°, 360.0°);默认值是 0.0° (Qt 4.6)
<code>scale()</code>	返回该项的缩放因子;默认值是 1.0 即不缩放 (Qt 4.6)
<code>scene()</code>	如果该项被添加到场景中,则返回它所属的场景
<code>sceneBoundingRect()</code>	返回该项在场景坐标系中的边界矩形;参见 <code>boundingRect()</code>
<code>scenePos()</code>	返回该项在场景坐标系中的位置——对于没有父项的项,该值与 <code>pos()</code> 返回的值相同
<code>setFlag(GraphicsItemFlag, bool)</code>	根据传入的布尔值设置标志的开/关 (默认是开)
<code>setFlags(GraphicsItemFlags)</code>	设置打开位或 (OR-ed) 标志;参见 <code>QGraphicsItem::GraphicsItemFlag</code> 枚举值

表 11.3 QGraphicsItem API(部分经过挑选的方法) #3

方 法	说 明
<code>setGraphicsEffect(QGraphicsEffect *)</code>	为该项设置传入的图形特效 (删除原有特效);特效包括 <code>QGraphicsBlurEffect</code> , <code>QGraphicsDropShadowEffect</code> 和 <code>QGraphicsOpacityEffect</code> (Qt 4.6)
<code>setGroup(QGraphicsItemGroup *)</code>	把该项添加到指定的组

(续表)

方 法	说 明
setParentItem(QGraphicsItem *)	设置(或改变)该项的父项为给定的项
setPos(QPointF)	设置该项的父项坐标系的坐标;另外还有一个接收两个 qreal 类型参数的重载函数
setRotation(qreal)	设置该项的旋转为给定角度值(-360.0° , 360.0°)(Qt 4.6)
setScale(qreal)	缩放该项;1.0 即不缩放(Qt 4.6)
setSelected(bool)	根据布尔值选中或取消选中该项
setToolTip(QString)	设置该项的工具提示(tooltip)
setTransform(QTransform, bool)	设置该项的变换矩形为指定值;如果布尔值为 true 则把指定值与原有变换矩形合并(Qt 4.3);另外还有一个颇为不同的 setTransformations() 函数
setVisible(bool)	根据给定的布尔值显示或者隐藏该项
setX(qreal)	设置该项在它的父项坐标系内的 x 坐标位置(Qt 4.6)
setY(qreal)	设置该项在它的父项坐标系内的 y 坐标位置(Qt 4.6)
setZValue(qreal)	设置该项的 z 值
shape()	重新实现这个函数以返回描述该项准确形状的绘制路径;参见 boundingRect() 和 paint() 函数
show()	显示该项;参见 hide() 和 setVisible() 函数
toolTip()	返回该项的工具提示
transform()	返回该项的变换矩形;另外还有一个 transformations() 函数
type()	把该项的 QGraphicsItem::Type 作为整型返回;自定义的 QGraphicsItem 子类通常需要重新实现这个函数并提供一个 Type 枚举类型

表 11.4 QGraphicsItem API(部分经过挑选的函数) #4

方 法	说 明
update()	安排执行处理该项的绘制事件
x()	返回该项在它的父项坐标系的 x 坐标
y()	返回该项在它的父项坐标系的 y 坐标
zValue()	返回该项的 z 值

表 11.5 Qt::ItemSelectionMode 枚举值

枚 举 值	说 明
Qt::ContainsItemShape	选择的项完全在选择区域中
Qt::IntersectsItemShape	选择的项在选择区域中或者与选择区域相交
Qt::ContainsItemBoundingRect	选择的项的边界矩形完全在选择区域中
Qt::IntersectsItemBoundingRect	选择的项的边界矩形在选择区域中或者与选择区域相交

表 11.6 QGraphicsItem::GraphicsItemFlag 枚举值#1

枚 举 值	说 明
QGraphicsItem::ItemAcceptsInputMethod	该项支持输入法(Qt 4.6)
QGraphicsItem::ItemClipsChildrenToShape	该项按自己的形状剪切它的所有子项(递归的)(Qt 4.3)
QGraphicsItem::ItemClipsToShape	该项按自己的形状剪切,不管它如何绘制,也不能接收在它的形状之外的事件(如鼠标点击)(Qt 4.3)
QGraphicsItem::ItemDoesntPropagateOpacityToChildren	该项的不透明性不会传递给它的子项(Qt 4.5)
QGraphicsItem::ItemHasNoContents	该项不绘制任何内容(Qt 4.6)
QGraphicsItem::ItemIgnoresParentOpacity	该项的不透明性仅为设置值,不合并父项的设定(Qt 4.5)

表 11.7 QGraphicsItem::GraphicsItemFlag 枚举值#2

枚举值	说明
<code>QGraphicsItem::ItemIgnores-Transformations</code>	该项忽略应用到它的父项的变换(虽然它的位置仍然与父项关联);对作为文本标签的项有用(Qt 4.3)
<code>QGraphicsItem::ItemIsFocusable</code>	该项接受键盘按下事件
<code>QGraphicsItem::ItemIsMovable</code>	该项(及其递归子项)可以通过点击和拖动移动
<code>QGraphicsItem::ItemIsPanel</code>	该项是一个面板(Qt 4.6);关于面板的更多信息参见在线文档
<code>QGraphicsItem::ItemIsSelectable</code>	该项可通过点击、框选拖动或者在 <code>QGraphicsScene::setSelectionArea()</code> 调用所影响的区域内被选中
<code>QGraphicsItem::ItemNegativeZStacks-BehindParent</code>	如果该项的 <i>z</i> 值为负,则自动放在父项的后方
<code>QGraphicsItem::ItemSendsGeometryChanges</code>	该项在位置或变换发生改变时调用 <code>itemChanged()</code> (Qt 4.6);参见“Qt 4.6 图形/视图行为的变化的”阴影部分
<code>QGraphicsItem::ItemSendsScene-PositionChanges</code>	该项在位置发生改变时调用 <code>itemChanged()</code> (Qt 4.6)
<code>QGraphicsItem::ItemStackBehindParent</code>	该项放在父项的后方而不是前方(默认情况下);创建阴影时比较有用
<code>QGraphicsItem::ItemUsesExtended-StyleOption</code>	为项提供额外的 <code>QStyleOptionGraphicsItem</code> 属性

至此我们完成了培养皿应用程序和 QGraphicsItem API 的学习。例子源码中有些我们不太感兴趣的细节未涉及,如每次循环结束用一个单次触发的定时器发起下一次循环——不能用固定时间间隔的 QTimer,因为每次循环的计算耗时不尽相同。另外在程序暂停时我们让整个窗体略微透明,这个特效在 Windows 下能很好的工作。

下一章将学习用更传统的方式使用图形/视图架构的例子程序,学习更多创建自定义图形项的例子,并涵盖如何把场景存入文件和如何从文件载入场景,以及如何操作场景中的项,如变换、复制、剪切和粘贴项。



第 12 章 创建图形/视图场景

- 场景、项和动作
- 增强 QGraphicsView 的功能
- 创建可停靠的工具箱窗口部件
- 创建自定义图形项

本章我们将介绍一个使用 Qt 的图形/视图架构的传统应用程序。该程序是基本的绘图应用，展示如何创建各种自定义项以及如何用自定义的文件格式保存和载入图形项。程序的实现还展示了如何提供用户向场景中添加项、修改项的属性（包括选中的项的组属性）和删除项的功能。本例和前一章的例子综合起来涵盖了图形/视图架构提供的许多功能特征——但这并不是全部。即便如此，本章和前一章也足以学习其他内容和开发自己的图形/视图应用程序提供坚实的基础。顺便提一句，正如上一章讲到的，我们将在第 13 章中重新回顾一下这两个示例程序，并将使用 Qt 4.6 专有的特性来创建示例程序的修改版本。

本章中我们将重新观察 Page Designer 程序 (pagedesigner1)。该例子是迄今为止书中介绍的最为“完整”的例子，用了将近 3300 行代码。但 Page Designer 仍然仅仅是一个骨架程序，缺少很多有用的功能。然而对于展示图形/视图的功能这一主要目的来说，这个示例足以胜任。图 12.1 展示了该应用程序。

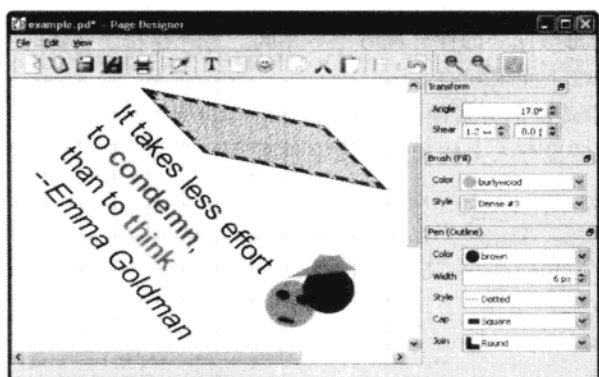


图 12.1 Page Designer 应用程序

Page Designer 是一个标准的主窗口风格的应用程序，使用了可停靠的窗口部件显示变换 (transformation)、设置画刷 (填充) 和设置画笔 (轮廓线) 三个“工具箱”。该应用程序具有所有标准的“文件”菜单选项：“新建”、“打开…”、“保存”、“另存为…”、“导出…” (以图片格式或 SVG 格式)、“打印…”和“退出”。特别是程序有一个“编辑”菜单与图形/视图编程相关，提供了“编辑选中项…” [如果项有自定义 edit() 槽函数则调用该函数，如弹出该项特定的上下文菜单 (context menu) 或对话框]、“添加文字…” [即添加一个富文本 (rich text) 项]、“添加方框” (即添加一个可改变大小的矩形) 和“添加表情符号” (即添加程序特定的、非矩形的自定义项)。另外，“编辑”菜单提供了常见的“剪切”、“复制”和“粘贴”选项，还有一个“对齐方式”子菜单 (用于排列两个以上选中的相关项) 和实用的“清除变换”选项，它重置选中项的旋转 (rotation) 和修剪 (shear) 属性为 0。程序还有一个“查看”菜单，用户通过该菜单可以放大、缩小和显示、隐藏参考网格 (guideline grid)。大多数菜单选项还可以通过工具栏按钮访问。

尽管程序仅提供了三种图形项(文字、方框和表情符号),但这几种已经可以代表绘图程序所需要的各种类型的图形项。例如,方框项相关的代码可以看做是添加任何规则形状图形项的例子,而表情符号相关的代码则可看做是添加不规则形状图形项的例子。

当选中某项时(如通过鼠标点击),工具箱被更新,显示出该项的角度、修剪、画刷和画笔的设置。Page Designer 还支持一些针对一组项的操作(QGraphicsView 类支持通过按下 Ctrl 键并点击项来选择项目,在 Mac OS X 上为 ⌘ + 鼠标点击。另外还可以(我们后面会介绍)打开橡皮筋(rubber band)选择功能,这样橡皮筋的矩形涵盖或接触到的项会被选中)。例如,所有选中的项可以被剪切、复制、删除或按相互关系对齐排列。类似地,清除变换、设置变换和设置画刷、画笔也应用到所选的一个或多个项上。

本章将概览程序的基本架构看看它是如何工作的(包括保存场景和载入场景),然后我们将深入了解图形/视图相关的各方面内容。首先从程序的主窗口开始。

12.1 场景、项和动作

应用程序的主窗口为每个菜单动作(文件、编辑和查看)各添加了一个私有的槽,另外还有很多私有的帮助成员函数。这里从主窗口的私有数据(但不包括 QAction)看起,作为基本的背景介绍。

```
private:
...
TransformWidget *transformWidget;
BrushWidget *brushWidget;
PenWidget *penWidget;

QPrinter *printer;
QGraphicsScene *scene;
GraphicsView *view;
QGraphicsItemGroup *gridGroup;
QPoint previousPoint;
int addOffset;
int pasteOffset;
};
```

三个自定义的窗口部件是可停靠窗口容纳的工具箱。我们保存部件的指针,因为每次向场景中添加新项的时候都需要把该项与工具箱关联,这样当该项被选中时,它的属性(如画刷和画笔)会在工具箱中显示出来。我们将在后面的小节介绍其中的一个窗口部件。

保存一个 QPrinter 的指针,这样每次用户打印时可以方便地获取上次打印的设置。当 QPrinter 创建时使用切合实际的初始默认值,如在美国地区使用美式信纸尺寸,而在欧洲则使用 A4 尺寸。

我们选择把参考网格创建为图形项,而没有把它设置为背景,这仅仅是为了演示如何使用 QGraphicsItemGroup,以及如何有选择性地保存、打印和导出项。

我们将在后面讲到 previousPoint、addOffset 和 pasteOffset 的代码时再讨论这几个变量。

这里没有保存文件名字符串——通过调用 setWindowFilePath() 函数保存文件名。这个方法以适合当前平台的方式把文件名放置到标题栏上(只要我们不用 setWindowTitle() 设置标题栏)(比如仅显示文件名而不包含路径)外加程序名。为了让这个函数符合我们的要求,需要在使用它之前(通常在 main() 函数中)调用 QApplication::setApplicationName()。当需要文件名时可以调用 windowFilePath() 函数获取——对于新文件则会返回“Unnamed”,假定我们在 fileNew() 函数中设置了这个字符串。

主窗口的其他部分将在下面的小节中介绍代码(有趣的部分)时呈现,下面先从主窗口的构造函数和几个帮助函数开始。

12.1.1 创建主窗口

对于大多数小型到中型(以程序大小论)的 Qt 应用程序而言,主窗口是它的核心。通常主窗口的构造函数是创建用户界面的地方——对外观和它的行为来说皆是如此。

```
const int OffsetIncrement = 5;
const QString ShowGrid("ShowGrid");
const QString MostRecentFile("MostRecentFile");

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), gridGroup(0), addOffset(OffsetIncrement),
      pasteOffset(OffsetIncrement)
{
    printer = new QPrinter(QPrinter::HighResolution);

    createSceneAndView();
    createActions();
    createMenusAndToolBars();
    createDockWidgets();
    createConnections();

    QSettings settings;
    viewShowGridAction->setChecked(
        settings.value(ShowGrid, true).toBool());
    QString filename = settings.value(MostRecentFile).toString();
    if (filename.isEmpty() || filename == tr("Unnamed"))
        QTimer::singleShot(0, this, SLOT(fileNew()));
    else {
        setWindowFilePath(filename);
        QTimer::singleShot(0, this, SLOT(loadFile()));
    }
}
```

构造函数先设置几个变量(将在后面用到它们的时候再讨论),然后创建一个打印机——打印机的页面大小在重新实现的 `sizeHint()` 函数(这里未做介绍)中用做主窗口初始大小的基础。用户界面其他部分的设置分散到各个不同的帮助函数中,这样的风格我们应该已经很熟悉了。`createActions()`、`createMenusAndToolBars()` 和 `createConnections()` 函数都遵循同样的模式,在前面已经见过多次,所以在这里就不列出代码了(与往常一样,源码可以在书的例子中找到,本例在 `pagedesigner1` 子目录中)。

我们先根据上次程序运行时设置的参考网格显示与否来显示或者隐藏网格——如果程序是第一次运行就默认显示网格。我们将在后面介绍私有的自定义槽 `viewShowGrid()` (连接到 `viewShowGridAction` 的 `triggered()` 信号)。

在构造函数的最后我们获取上次运行时编辑的文件名。如果该文件名为空(程序第一次运行)或者是“Unnamed”(上次编辑的页面设计未保存),就调用 `fileNew()` 槽(这里未做介绍),这样用户可以直接开始绘制。否则我们载入文件。与往常一样,使用了单次触发的定时器(single shot timer)以在构造完成后调用 `fileNew()` 和 `loadFile()` 函数。

关于 `ShowGrid` 和 `MostRecentFile` 两个变量值的保存,则放在了重新实现的函数 `closeEvent()` 中(这里没有展示)。

```
void MainWindow::createSceneAndView()
{
    view = new GraphicsView;
    scene = new QGraphicsScene(this);
    QSize pageSize = printer->paperSize(QPrinter::Point).toSize();
    scene->setSceneRect(0, 0, pageSize.width(), pageSize.height());
    view->setScene(scene);
    setCentralWidget(view);
}
```

视图、场景和应用程序中心窗口部件的创建和设置都十分简单明了。场景的尺寸和页面的大小成正比,实际上是将磅值(1/72")映射为像素(`paperSize()`函数返回 `QSizeF` 类型,我们用 `QSizeF::toSize()` 将它转换为 `QSize` 类型)。只有一点需要说明的是,这里使用了一个自定义的 `GraphicsView` 类(`QGraphicsView` 的子类),提供缩放和鼠标滑轮的支持。我们将在后面介绍这个简单的子类。

```
void MainWindow::createDockWidgets()
{
    setDockOptions(QMainWindow::AnimatedDocks);
    QDockWidget::DockWidgetFeatures features =
        QDockWidget::DockWidgetMovable |
        QDockWidget::DockWidgetFloatable;

    transformWidget = new TransformWidget;
    QDockWidget *transformDockWidget = new QDockWidget(
        tr("Transform"), this);
    transformDockWidget->setFeatures(features);
    transformDockWidget->setWidget(transformWidget);
    addDockWidget(Qt::RightDockWidgetArea, transformDockWidget);
    ...
}
```

这个函数是三个工具箱部件创建和加入到可停靠窗口部件的地方。这里只显示第一个部件的代码,因为除了具体的工具箱部件不同以及添加到的可停靠窗口部件不同之外,其他代码完全一样。

先设置停靠选项, `QMainWindow::AnimatedDocks` 选项表示当用户把可停靠窗口部件拖曳到可停靠区域时,停靠区域为它创建一个空白区域显示它可能停靠的位置,如有需要则移动其他可停靠部件让出该位置。这样用户更容易看到可能发生的改变。

默认停靠选项是 `QMainWindow::AnimatedDocks | QMainWindow::AllowTabbedDocks`, 所以我们设置前一个选项也就是禁用了后一个。如果允许标签式停靠(`tabbed dock`), 用户可以把可停靠部件放在其他可停靠部件之间(通常的表现), 或者放在另一个可停靠部件之上, 这样 Qt 就把放下的可停靠部件和下面的可停靠部件放在一个标签部件里, 放下的窗口部件可见(在最上面的标签页), 这样用户就可以点击标签切换部件。默认情况下标签出现在可停靠部件的底部, 但可以通过调用 `QMainWindow::setTabPosition()` 函数来修改。显而易见, 当可停靠部件数量较多时, 或者当某个部件占据较大空间时, 这个选项很有用。

另外还有两个停靠选项可以用(两者互斥)。一个是 `QMainWindow::ForceTabbedDocks`——强制所有可停靠部件停靠为标签页面, 缺点是每个停靠区域只有一个可停靠部件可见。另一个选项是 `QMainWindow::AllowNestedDocks`——设置这个项以后每个停靠区域可以分割, 以停靠多行或多列的窗口部件。该选项的缺点是用户界面变得更复杂, 用户不易操作。应尽量避免使用这两个选项。

每个可停靠部件都设置了两个属性: `QDockWidget::DockWidgetMovable` (表示用户可以把可停靠部件从一个停靠区域拖曳到另一个停靠区域) 和 `QDockWidget::DockWidgetFloatable` [表示用户可以把可停靠部件拖曳离原停靠区域, 使它成为浮动的独立 (`free-standing`) 窗口部件]。假如我们想提供给用户关闭(隐藏)可停靠部件的选项, 则可以设置 `QDockWidget::DockWidgetClosable`。另一个支持的功能 `QDockWidget::DockWidgetVerticalTitleBar` 可以减少可停靠部件纵向占用的空间, 对于左、右停靠区域的可停靠部件来说极其有用。图 12.2 展示了一个带有垂直标题栏的可停靠部件和一个拽出停靠区域并显示为独立控制的浮动窗口的可停靠部件。

我们未列出用于创建主窗口的动作、菜单和工具栏的代码, 因为这些都是标准代码, 但在后面的小节中我们将浏览实现了程序高层图形/视图相关行为的函数, 从保存、载入场景的函数开始。

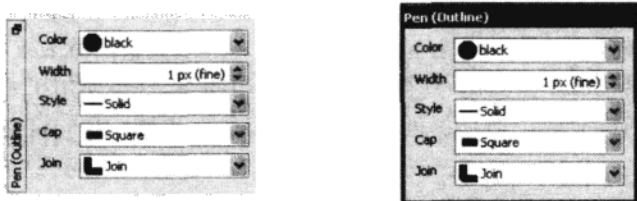


图 12.2 Page Designer 中带垂直标题栏和独立浮动的画笔可停靠窗口部件

12.1.2 保存、载入、打印和导出场景

Page Designer 应用程序具备常规的“保存”、“打开”、“打印…”和“导出…”动作,与之对应为 `fileSave()`、`fileOpen()`、`filePrint()` 和 `fileExport()` 槽。我们将略过保存和载入的具体细节(与前面例子中用到的非常相似),仅着眼于图形/视图相关的方法和帮助函数。

12.1.2.1 保存场景

`fileSave()` 槽中程序以二进制方式打开了一个 `QFile` 对象,然后打开 `QDataStream` 写入文件。页面设计器的幻数(magic number)和文件格式版本号(均为整型值)被写入数据流,然后设置数据流的版本(本例为 Qt 4.5)。意思是程序不能基于更早的 Qt 版本编译,这样保证数据可以被任何 Qt 4.x 版本($x \geq 5$)读取。接下来调用 `MainWindow::writeItems()` 函数,传入两个参数——数据流和场景中所有项的列表。

我们用 `QGraphicsScene::items()` 函数提供第二个参数,该函数返回场景的图形项列表。有好几个 `items()` 的重载函数,包括接收排序方式作为参数的、返回与某个点相交的项、返回在某矩形、多边形、绘制路径内或与矩形、多边形、绘制路径相交的项。本例中返回的项包含我们不需要也不想保存的参考网格项,但这不算问题,我们稍后会看到。

```
void MainWindow::writeItems(QDataStream &out,
                           const QList<QGraphicsItem*> &items)
{
    foreach (QGraphicsItem *item, items) {
        if (item == gridGroup || item->group() == gridGroup)
            continue;
        qint32 type = static_cast<qint32>(item->type());
        out << type;
        switch (type) {
            case BoxItemType:
                out << *static_cast<BoxItem*>(item); break;
            case SmileyItemType:
                out << *static_cast<SmileyItem*>(item); break;
            case TextItemType:
                out << *static_cast<TextItem*>(item); break;
            default: Q_ASSERT(false);
        }
    }
}
```

这个函数遍历传入的列表中的所有图形项,对于每个项先检查它是不是一个 `gridGroup` (`QGraphicsItemGroup *` 类型),或者是否属于参考网格项所属的 `gridGroup`,如果是就略过这个项,因为我们不想保存参考网格。函数把每个其他的项的 `QGraphicsItem::Type` (由重新实现的 `QGraphicsItem::type()` 函数返回的、唯一标志图形项的类型的常整数)写入文件,然后是图形项本身。主窗口的 `writeItems()` 函数并不知道如何保存图形项,它依赖于图形项提供 `<<()` 操作符函数操作 `QDataStream` (当然所有的图形项都提供了该操作符函数,我们将在讲到图形项的时候看到)。

将 `writelnItems()` 函数分离出 `fileSave()` 函数的另一个好处是,可以在剪切和复制项时重用该函数——在后面的小节中将会涉及这部分内容。

调用 `writelnItems()` 函数,用尽可能简单的方法(仅仅保存了图形项的序列)保存场景。更精妙的程序可能会在保存项本身的同时保存一些元数据(meta-data)。现在我们了解了场景是如何保存的,下面将浏览它如何载入、如何打印以及如何把场景导出为标准的图像(pixmap)格式和矢量图格式。

12.1.2.2 载入场景

在 `fileOpen()` 槽中程序弹出对话框让用户提供一个文件名,用户选中文件名后为该文件创建一个 `QFile`。然后创建 `QDataStream` 读取文件的内容,读入幻数和文件格式版本号。如果出现任何问题就弹出对话框通知用户,否则设置数据流为 Qt 4.5 以匹配保存时使用的版本,然后清空场景中已存在的项,读入文件中保存的项,填充到场景中。

有些遗憾的是,`readItems()` 函数(与刚刚介绍的 `writelnItems()` 函数和后面要介绍的 `editAddItem()` 函数类似)必须根据特定的已知类型操作,也就是说,如果我们添加了新的类型,这三个函数都必须修改以计入添加的新类型。然而对于这个问题貌似没有什么完美的解决方法。好消息是所有其他的成员函数都使用 Qt 的属性系统(所有的自定义项的类型都继承 `QObject`),意味着即使我们添加了新的类型,这些函数仍然可以不经修改就能正确工作,当然前提是新的项类型必须也继承自 `QObject`。

```
void MainWindow::clear()
{
    scene->clear();
    gridGroup = 0;
    viewShowGrid(viewShowGridAction->isChecked());
}
```

`QGraphicsScene::clear()` 函数删除所有的图形项,包括用于参考网格的项,所以我们必须确保设置 `gridGroup` 为 `null`(因为它已被删除)。当然,如果参考网格是用背景画刷创建的,或者通过重新实现 `QGraphicsView::drawBackground()` 函数绘制的(在实践中我们会使用这两种方法之一),就不需要 `gridGroup` 了[对于页面设计器程序,对背景使用 `QGraphicsItemGroup` 完全是为了展示如何使用该类以及如何项集(set of item)中区分项]。然后,调用 `viewShowGrid()` 创建新的参考网格集合,根据 `viewShowGridAction` 的状态设置它的显示或隐藏。

```
void MainWindow::readItems(QDataStream &in, int offset, bool select)
{
    QSet<QGraphicsItem*> items;
    quint32 itemType;
    QGraphicsItem *item = 0;
    while (!in.atEnd()) {
        in >> itemType;
        switch (itemType) {
            case BoxItemType: {
                BoxItem *boxItem = new BoxItem(QRect(), scene);
                in >> *boxItem;
                connectItem(boxItem);
                item = boxItem;
                break;
            }
            ...
        }
        if (item) {
            item->moveBy(offset, offset);
            if (select)
                items << item;
        }
    }
}
```



```

        item = 0;
    }
}
if (select)
    selectItems(items);
else
    selectionChanged();
}

```

这个函数用于从一个文件上打开的 `QDataStream` 对象中读取场景的项。还用于把复制或剪切的项粘贴回场景中,这种情况下是用 `QByteArray` 上打开的数据流从剪贴板 (clipboard) 读取而不是从文件读取。`offset` (默认值为 0) 和 `select` (默认值为 `false`) 仅在粘贴时才用到。

函数读取数据流直至末尾,每个项读取两部分,先是项的 `QGraphicsItem::Type`,然后是项本身。读取表情符号和文本项的代码省略了,因为几乎和方框项的代码一模一样,不同之处是表情符号和文本项的构造函数接收 `QPoint()` 而不是 `QRect()`。所有的项都要和相关的工具箱部件相联,在 `connectItem()` 函数中完成,我们将在后面观察该函数。与 `writeItems()` 函数类似,`readItems()` 函数并不知道如何读取自定义的项类型,它依赖于项提供的 `>>()` 操作符函数操作于 `QDataStream` (当然所有的项都提供了该操作符)。

粘贴项时 `offset` 为非零值, `select` 为 `true`, 在这种情况下,项是从它当前的位置相对移动的,并添加到一组项中。最后如果 `select` 为真,则集合中的所有项都被选中;否则我们调用 `selectionChanged()` 方法确保工具箱部件显示出选中项的属性,因为最后一个读入的项将在它的构造函数中选中自己。

```

void MainWindow::connectItem(QObject *item)
{
    connect(item, SIGNAL(dirty()), this, SLOT(setDirty()));
    const QMetaObject *metaObject = item->metaObject();
    if (metaObject->indexOfProperty("brush") > -1)
        connect(brushWidget, SIGNAL(brushChanged(const QBrush&)),
                item, SLOT(setBrush(const QBrush&)));
    if (metaObject->indexOfProperty("pen") > -1)
        connect(penWidget, SIGNAL(penChanged(const QPen&)),
                item, SLOT(setPen(const QPen&)));
    if (metaObject->indexOfProperty("angle") > -1) {
        connect(transformWidget, SIGNAL(angleChanged(double)),
                item, SLOT(setAngle(double)));
        connect(transformWidget, SIGNAL(shearChanged(double, double)),
                item, SLOT(setShear(double, double)));
    }
}

```

在新项添加到场景中时,该函数被 `readItems()` 函数调用(比如当从场景文件中读入一个新项,或当一个新项被粘贴到场景中时)。它要求项必须是 `QObject` 的子类,这样我们才能利用 Qt 的属性系统来判断该建立哪个信号-槽的连接。这个方案的缺点是所有的自定义项类型都必须继承自 `QObject` 类,但优点是我们不必了解哪个具体的项类型被连接——这样一来,这个函数变得扩展性更强,因为不论添加多少个新的项类型,这个方法都不需要修改。

在页面设计器程序中,我们采用了传统方法,所有的项(除了参考网格项)都是 `QObject`,当它的状态发生重要改变时会发出一个自定义的 `dirty()` 信号。

针对每个感兴趣的属性,我们向项[具体来说,是元对象 (meta-object) 对应的 `QObject` 类的项]的元对象查询是否有该属性(返回的索引值为 -1 表示该属性不存在)。

方框项的属性有画刷、画笔、(旋转的)角度和水平/垂直修剪 (shear),我们为这些属性建立连接。在页面设计器程序中遵循常规做法,如果一个项有角度属性则它也会被修剪掉 (be sheared)。

我们可以容易地按照自己的意愿将属性分组,比如可以选择假定有画笔属性的项也有画刷属性,这样就不必对每个属性都进行检查了。

这些连接可以确保当用户在任意一个工具箱窗部件中更改了属性时(比如用户更改了画刷),这个工具箱项(事实上还包括所有连接到 BrushWidget 的 brushChanged() 信号的项)会接到更改通知。在介绍方框项时将会看到,通知只作用于选中的项。

可以在 readItems() 函数中建立这些连接,但我们选择把它提取出来,因为当用户添加新项到场景中时还要用到。

前面我们看到,在 readItems() 函数的最后,如果读入的项需要选中(当项是粘贴过来而不是从文件读入时就是这样),我们调用 selectItems() 函数完成选择动作。

```
void MainWindow::selectItems(const QSet<QGraphicsItem*> &items)
{
    scene->clearSelection();
    foreach (QGraphicsItem *item, items)
        item->setSelected(true);
    selectionChanged();
}
```

该函数取消所有场景中的项的选中状态,然后遍历所有的项,如果需要就选中项。接着它调用 selectionChanged() 函数,这样一来工具箱部件就被更新以显示出选中的项的属性。

12.1.2.3 打印和导出场景

除了保存和载入,我们还可以打印和导出场景为矢量图(SVG)格式以及任何一种 Qt 支持的图片格式。用 filePrint() 函数可以很容易实现打印。打印或者导出场景需要创建合适的 QPainter, 去除参考网格并将场景渲染到绘制对象(painter)。

```
const int StatusTimeout = AQP::MSecPerSecond * 30;

void MainWindow::filePrint()
{
    QPrintDialog dialog(printer);
    if (dialog.exec()) {
        {
            QPainter painter(printer);
            paintScene(&painter);
        }
        statusBar()->showMessage(tr("Printed %1")
                                .arg(windowFilePath()), StatusTimeout);
    }
}
```

打印对话框提供用户在打印之前修改打印相关设置的机会,我们传入 QPrinter 的指针,这样打印对话框打开时会用 QPrinter 的设置作为它的默认值。通过这种方式可以在打印之后保存打印设定(供下次打印使用)。如果用户接受对话框,就创建一个向打印机绘制的绘制对象(绘制到打印机页面),然后用自定义的 paintScene() 函数绘制场景。

这个函数与其他调用 paintScene() 的函数相比有个小小的不寻常之处,我们可以在一个范围(scope)内创建绘制对象和绘制场景(比如在一个大括号括起的代码段内)。这样做仅仅是为了便于确认绘制对象在使用完毕时被销毁,因为在析构时会释放它使用的所有资源。另一个不需要使用范围的可选方法是在绘制完毕时调用 QPainter::end() 函数。

windowFilePath() 函数返回的字符串是用 setWindowFilePath() 设置的文件名(包括路径),有可能是“Unnamed”。当新文件未被保存时,假设我们在 fileNew() 函数设置了这个名字的话(当然我们正是这么做的)。

```

void MainWindow::paintScene(QPainter *painter)
{
    bool showGrid = viewShowGridAction->isChecked();
    if (showGrid)
        viewShowGrid(false);
    QList<QGraphicsItem*> items = scene->selectedItems();
    scene->clearSelection();
    scene->render(painter);
    if (showGrid)
        viewShowGrid(true);
    foreach (QGraphicsItem *item, items)
        item->setSelected(true);
    selectionChanged();
}

```

渲染场景时我们不希望画出参考网格或者选择矩形,为了实现这个需求,我们隐藏所有的参考网格(调用 `viewShowGrid()`,这个函数将在后面介绍),同时保存选中项的列表然后清空选择。

准备工作完成后我们把场景绘制到绘制对象。可以选择把目的矩形(当绘制到已有的图片上时)和源矩形(仅绘制场景中的特定部分)传给 `QGraphicsScene::render()` 函数,如果源矩形和目的矩形都提供了,那么还可以传入另一个可选的、类型为 `Qt::AspectRatioMode` 的参数,当源和目的矩形尺寸不同时这个参数有意义。

绘制完毕之后根据参考网格原先的显示状态显示网格,并根据原来项的选中状态重新选择项,这样场景就恢复到导出之前的状态了。我们还调用 `selectionChanged()` 函数让工具箱窗口部件显示出选中项的属性。

当导出场景时,对绘制对象的设定与第 10 章中导出 `QTextDocuments` 时的设定非常相似。为了完整起见,我们把这两个导出函数都列出来,两个函数都需要用到刚刚介绍的自定义的 `paintScene()` 函数。`fileExport()` 槽弹出一个文件保存对话框,根据用户输入的文件名的后缀选择要调用的导出方法。

```

void MainWindow::exportSvg(const QString &filename)
{
    QSvgGenerator svg;
    svg.setFileName(filename);
    svg.setSize(printer->paperSize(QPrinter::Point).toSize());
    {
        QPainter painter(&svg);
        paintScene(&painter);
    }
    statusBar()->showMessage(tr("Exported %1").arg(filename),
                             StatusTimeout);
}

```

我们要在应用程序的 .pro 文件中加入 `QT += svg` 才能使用 Qt 的 SVG 功能。首先创建一个 SVG 生成器(SVG generator)对象,传入要写入的文件名。设置它的页面大小为 Page Designer 的页面大小,创建绘制到 SVG 生成器的绘制对象,然后调用 `paintScene()` 帮助函数完成这个工作。通过把 `QPrinter` 作为绘制设备,可以用类似的方法把场景绘制到一个 PDF 文件中,就像第 10 章中介绍的那样,这将作为课后练习留给读者。

```

void MainWindow::exportImage(const QString &filename)
{
    QImage image(printer->paperSize(QPrinter::Point).toSize(),
                  QImage::Format_ARGB32);
    {

```

```

    QPainter painter(&image);
    painter.setRenderHints(QPainter::Antialiasing|
                           QPainter::TextAntialiasing);
    paintScene(&painter);
}
if (image.save(filename))
    statusBar()->showMessage(tr("Exported %1").arg(filename),
                             StatusTimeout);
else
    AQP::warning(this, tr("Error"), tr("Failed to export: %1")
                 .arg(filename));
}

```

导出图片文件的做法与导出 SVG 图像十分相似,除了打开反走样(antialiasing)[因为常规做法是在屏幕(on-screen)和图片(pixmap images)中使用反走样,而在打印或矢量图(vector images)中则不使用]。另外 QImage::save() 返回一个布尔值表示成功与否,我们也用到了该返回值。

至此我们了解了如何保存、载入、打印和导出场景,已经准备好学习如何把各个项添加到场景以及如何控制项——如复制、剪切和粘贴以及对齐。

12.1.3 控制图形项

Page Designer 支持添加项、控制单独的项(如某个选中的项),某些操作还可以控制所有选中的项。例如,如果用户调用“编辑选中的项...”动作,下面的槽就会被调用:

```

void MainWindow::editSelectedItem()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    if (items.count() != 1)
        return;
    if (QObject *item = dynamic_cast<QObject*>(items.at(0))) {
        const QMetaObject *metaObject = item->metaObject();
        metaObject->invokeMethod(item, "edit", Qt::DirectConnection);
    }
}

```

在同一时刻,我们只允许一个项被编辑,所以首先检查是否只有一个选中的项,如果只有一项被选中,就获取该项相关的元对象,并尝试调用该项的 edit() 槽。如果槽的名字存在并可被调用,那么 QMetaObject::invokeMethod() 函数返回 true,否则该函数什么都不做直接返回 false。Qt::DirectConnection 参数告诉 Qt 立即调用槽而不需等待事件循环空闲时调度。我们在前面的章节中讨论了 invokeMethod() 函数。这段代码表明 editSelectedItem() 函数不需要了解关于选中的项的类型信息,而且即使添加了新的类型也可以正确工作,不论新的项类型是否有 edit() 槽——只要该类型从 QObject 派生即可。

对于 Page Designer 程序来说,我们的 BoxItem(方框项)没有 edit() 槽,所以这个函数仅仅是无害的,什么都不做。而 SmileItem(表情符号项)和 TextItem(文本项)都有 edit() 槽,对表情符号会弹出一个上下文菜单,对文本项则会弹出编辑对话框。所以假如要允许对每个选中的项而不是仅对一项调用编辑功能,可能会出现弹出几十个甚至上百个菜单和对话框的情况。

这里(和别处)用到的 dynamic_cast<>() 意味着 Page Designer 依赖于 RTTI(Run Time Type Information 运行时类型信息)。大多数现代桌面系统的编译器都支持并默认打开这个功能,但对嵌入式设备的编译器来说就未必打开甚至可能不支持该功能了。替代方法是使用不依赖 RTTI 的 qobject_cast<>() 函数,但不巧的是,本例不能使用该函数,因为它不能接收 QGraphicsItem 指针作为参数。但还是有可能使用 qobject_cast<>() 的,只要所有的图形项都从 QGraphicsObject 类派生,至少在引入 QGraphicsItem::toGraphicsObject() 函数的 Qt 4.6 可以用如下代码转换:QObject *

`object = qobject_cast<QObject*>(item->toGraphicsObject())`。在 Page Designer 例子中,这个转换可以应用到表情符号项和文本项上,两者都是从 `QGraphicsObject` 派生的,而方框项并没有从 `QGraphicsObject` 类派生,所以不能应用这个方法,但我们稍后会看到如何绕过这个问题。

一种可以避免使用 `dynamic_cast<>()` 也避免使用 Qt 的元对象系统(比如可能不想让所有的自定义项都从 `QObject` 派生)的方法是:使用 `qgraphicsitem_cast<>()`。这样做的问题在于必须在代码中写死(hard-code)具体的项的类型,当然在写入、读取或添加项的时候都要这么做(固定写死项的类型)。要使用 `qgraphicsitem_cast<>()` 必须替换掉 `editSelectedItem()` 函数的第二个 if 语句,相关的代码块修改如下:

```
QGraphicsItem *item = items.at(0);
if (TextItem *textItem = qgraphicsitem_cast<TextItem*>(item))
    textItem->edit();
else if (SmileyItem *smileyItem =
    qgraphicsitem_cast<SmileyItem*>(item))
    smileyItem->edit();
```

我们把选中项作为 `QGraphicsItem` 而不是 `QObject` 获取它的指针,另外要注意这里没有提到 `BoxItem`,因为方框项没有 `edit()` 槽函数,所以可以忽略不计。假设我们添加 `BoxItem::edit()` 槽,那么必须要在上面的代码中添加另一个 else if 语句——但对于使用 Qt 的元对象系统的代码版本,该函数完全不需要修改。

对于 `pagedesigner2` 版本(将在第 13 章讨论,该例子需要 Qt 4.6)我们不用 `dynamic_cast<>()`,而是使用一个函数 `qObjectFrom()`。该函数接收一个 `QGraphicsItem` 指针返回 `QObject` 类型的指针。

```
QObject *qObjectFrom(QGraphicsItem *item)
{
    if (!item)
        return 0;
    // Types not inheriting QGraphicsObject must be handled explicitly
    if (item->type() == BoxItemType)
        return qobject_cast<QObject*>(static_cast<BoxItem*>(item));
    // Types inheriting QGraphicsObject can be handled generically
    return item->toGraphicsObject();
}
```

对于没有从 `QGraphicsObject` 派生的项,必须把它强制转换为实际的 `QGraphicsItem` 类(也必须继承自 `QObject`),然后再转换为 `QObject`。但对于从 `QGraphicsObject` 派生的项就可以简单地用 `QGraphicsItem::toGraphicsObject()` 返回需要的 `QObject` 指针。

有了这个函数,Qt 4.6 版本的 Page Designer 程序(使用 `#ifdef`)把每个用到 `dynamic_cast<QObject*>(item)` 的地方替换为 `qObjectFrom(item)`。这个方法的不足之处在于,如果添加新的(从 `QObject` 派生的)项的类型不是 `QGraphicsObject` 的子类,就必须修改 `qObjectFrom()` 函数显式地(explicitly)加入新类型;但如果使用 `dynamic_cast<>()` 函数,代码就不需要修改。

12.1.3.1 添加项

应用程序的用户界面为每个支持的项的类型提供了“添加”动作(action)。每个动作将相关的项的类型枚举值作为用户数据(user data),并都连接到同一个 `editAddItem()` 槽。

```
void MainWindow::editAddItem()
{
    QAction *action = qobject_cast<QAction*>(sender());
    if (!action)
        return;
    QObject *item = 0;
    int type = action->data().toInt();
    if (type == BoxItemType)
```

```

        item = new BoxItem(QRect(position(), QSize(90, 30)), scene);
    else if (type == SmileyItemType)
        item = new SmileyItem(position(), scene);
    else if (type == TextItemType) {
        TextItemDialog dialog(0, position(), scene, this);
        if (dialog.exec())
            item = dialog.textItem();
    }
    if (item) {
        connectItem(item);
        setDirty(true);
    }
}

```

用户请求添加任何一种项时都会调用到该槽(我们在前面讨论了如何使用 `QObject::sender()` 和其他替换方案)。如果请求的是方框项,该函数赋予该项默认的大小和 `position()` 函数返回的位置信息,我们将在后面看到这个函数。对于文本项,与简单地对方框和表情符号创建默认项不同的是,我们选择弹出一个对话框,这样用户可以在第一时间设置文本项的文本内容。`TextItemDialog`(代码未列出,但截图可见图 12.3)是一个典型的添加/编辑对话框,第一个参数是等待编辑的项(通过调用项的 `edit()` 槽调出对话框时)或 0(如本例),如果传入 0 就表示需要创建一个新项(如果用户点击对话框上的 OK 按钮)——如果用户选择接受对话框,我们用 `TextItemDialog::textItem()` 函数获取最近创建的 `TextItem`。

当创建好请求的项,我们把它连接到对应的工具箱窗口部件,这样用户就可以修改项的属性——以方框项为例,它的画刷、画笔和变换。我们在前面介绍了 `connectItem()` 方法。由于添加新项明显改变了场景,当然需要调用 `setDirty()`。

```

const int OffsetIncrement = 5;

QPoint MainWindow::position()
{
    QPoint point = mapFromGlobal(QCursor::pos());
    if (!view->geometry().contains(point)) {
        point = previousPoint.isNull()
            ? view->pos() + QPoint(10, 10) : previousPoint;
    }
    if (!previousPoint.isNull() && point == previousPoint) {
        point += QPoint(addOffset, addOffset);
        addOffset += OffsetIncrement;
    }
    else {
        addOffset = OffsetIncrement;
        previousPoint = point;
    }
    return view->mapToScene(point - view->pos()).toPoint();
}

```

这个函数用于给新添加的项提供合适的位置。首先创建一个鼠标所在位置的 `QPoint`,如果该点没有在视图内,就把点设置到视图中——或如果有上一个项添加的位置点的话就设置为上一个点的位置。然后,如果新点与上一个项添加的位置点相同,就给这个点增加一个位移,把它稍微向右下方移动一点,并增加位移值的增量。否则,如果新的点与上一个点位置不同,就重置增量的值为原始值并把上一个点设置为当前的点。在函数末尾返回点在场景中的位移,以场景坐标系表示。这样就确保重复添加项时,每个项都以稍微偏右下的位移(而不是叠加在一起)添加进来,让用户能看得更清楚。

至此我们已经学习了单独的项如何调用它们的 `edit()` 函数(如果有这个函数的话),以及用户是如何添加这些项的,下面将介绍 `Page Designer` 是如何支持复制、剪切和粘贴的。

12.1.3.2 复制、剪切和粘贴项

为了支持复制、剪切和粘贴,我们用到了系统剪贴板。这样的好处是别的程序可以访问到我们复制或剪切的项,Page Designer 能粘贴别的程序放入剪贴板的项(如果我们可以理解别的数据格式的话)。使用剪贴板保存(而不是以类似于用一个私有 `QByteArray` 成员变量的保存)我们复制或剪切的项有一点不好的地方,那就是如果用户的上下文切换到另一个应用程序,并发生了复制或剪切操作,我们的项就从剪贴板删除了。解决方案是提供对 `undo/redo` 功能的支持,这部分不做介绍,但我们将在本章的末尾再次提及。

除了使用剪贴板之外我们还维护一个私有成员变量 `pasteOffset`(整型),用于确保当项被粘贴时,从原始位置稍微移动一点,这样用户可以更清楚地看到发生了粘贴操作。

```
void MainWindow::editCopy()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    if (items.isEmpty())
        return;
    pasteOffset = OffsetIncrement;
    copyItems(items);
    updateUi();
}
```

用户调用“复制”动作时,该槽获取选中项的列表,重置粘贴位移(`paste offset`)为初始值(5 个像素),然后调用 `copyItems()` 帮助函数把选中项复制到剪贴板。这样这些项就可以粘贴回去;但在粘贴之前,如果在当前程序或其他程序中复制(或剪切)了其他内容,或用户退出程序,那么这些项就丢失了(被覆盖了或删除了)。函数最后调用 `updateUi()` 确保“粘贴”动作使能(`enabled`)。

```
const QString MimeType = "application/vnd.qtrac.pagedesigner";

void MainWindow::copyItems(const QList<QGraphicsItem*> &items)
{
    QByteArray copiedItems;
    QDataStream out(&copiedItems, QIODevice::WriteOnly);
    writeItems(out, items);
    QMimeData *mimeTypeData = new QMimeData;
    mimeTypeData->setData(MimeType, copiedItems);
    QClipboard *clipboard = QApplication::clipboard();
    clipboard->setMimeData(mimeTypeData);
}
```

上面的函数创建一个空的 `QByteArray`,并使用前面看到过的 `writeItems()` 方法把要复制的项填充至其中。接下来在堆(heap)上创建一个新的 `QMimeData` 对象,传入描述用于标志数据格式的、自定义 MIME 类型的字节数组(byte array)。然后获取系统剪贴板的指针,把 MIME 数据存入剪贴板——剪贴板将获得数据的所有权(ownership),所以我们不必删除它。

```
void MainWindow::editCut()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    if (items.isEmpty())
        return;
    copyItems(items);
    QListIterator<QGraphicsItem*> i(items);
    while (i.hasNext()) {
        QScopedPointer<QGraphicsItem> item(i.next());
        scene->removeItem(item.data());
    }
    setDirty(true);
}
```


从场景中剪切项比复制数据复杂一些,因为我们既要复制项又要移除项。这个函数的开头和 `editCopy()` 函数相同,但完成复制后它将继续,从场景中移除(`remove`)并删掉(`delete`)每个复制完毕的项。

`QGraphicsScene::removeItem()` 函数从场景中移除给定的项(包括项的子孙,递归移除),然后把它的所有权传递给调用者。为了防止内存泄漏,一旦在一定范围内使用的指针超出范围,即每次循环结束,我们立刻删除移除的项(递归删除它的子孙项)^①。由于剪切项与单纯的复制项不同,会改变场景,所以在函数最后调用 `setDirty()`(`setDirty()` 槽调用 `updateUi()` 槽,所以剪切之后“粘贴”动作将使用)。

```
void MainWindow::editPaste()
{
    QClipboard *clipboard = QApplication::clipboard();
    const QMimeData *mimeTypeData = clipboard->mimeTypeData();
    if (!mimeTypeData)
        return;

    if (mimeTypeData->hasFormat(MimeType)) {
        QByteArray copiedItems = mimeTypeData->data(MimeType);
        QDataStream in(&copiedItems, QIODevice::ReadOnly);
        readItems(in, pasteOffset, true);
        pasteOffset += OffsetIncrement;
    }
    else if (mimeTypeData->hasHtml() || mimeTypeData->hasText()) {
        TextItem *textItem = new TextItem(position(), scene);
        connectItem(textItem);
        if (mimeTypeData->hasHtml())
            textItem->setHtml(mimeTypeData->html());
        else
            textItem->setPlainText(mimeTypeData->text());
    }
    else
        return;
    setDirty(true);
}
```

如果用户复制或剪切了项,那么可以用这个函数将项粘贴回场景中。另外我们还添加了从其他程序粘贴 HTML 和纯文本的功能。

首先获得一个系统剪贴板(`system clipboard`)的指针并获取保存剪贴板数据的 `QMimeData` 对象。下一步检查 MIME 数据是否是 Page Designer 的自定义格式。如果存在这样的数据,我们用 `QByteArray` 读取数据然后用前面介绍的 `readItems()` 函数将项填充到场景中。与从文件中读取项不同的是,我们提供了一个位移值,以确保粘贴的项不会正好粘贴到原有项之上(不然用户可能看不出这些项),同时这些项被选中,以便用户可以对它进行操作——比如将它作为一组移动或删除。之后增加粘贴的位移;这样保证用户再次粘贴同样的项时每次位移都相对于上次的位移,同样是为了让这些项保持对用户可见。

如果 MIME 数据不能提供给我们自定义的 Page Designer 格式,则检查它提供的是不是 HTML 或者纯文本,对这两种情况在场景中合适的位置(用 `position()` 函数)创建一个新的 `TextItem`,并将之连接到工具箱窗口部件,然后用对应的函数把 HTML 或纯文本加到文本项中(注意,应该按照我们倾向的顺序检查 MIME 格式,如最倾向的放在最前面。如下情况,如果剪贴板中是 HTML 文本, `QMimeData::hasText()` 和 `QMimeData::hasHtml()` 函数都可能返回 `true`,因为很多程序会向剪贴板中复制多种格式,而且很可能同时提供对 `text/html` 和 `text/plain` 格式的支持)。

① 在代码中我们用了 `#if QT_VERSION`, 这样代码可以在 Qt 4.5 编译,在这部分代码中使用普通的 `QGraphicsItem *` 为每个从场景中移除的项调用删除动作。

在最后,如果我们通过读取内容添加了项或者创建了新文本项,则调用 `setDirty()` 方法,因为粘贴操作明显改变了场景。

12.1.3.3 选中项的操作

我们将在本小节介绍 `editAlign()` 槽和它的帮助函数,观察一个典型的、操作两个及以上选中项的方法是如何实现的。另外还介绍当选择改变时如何处理。

用户界面允许用户采用两种方式对齐多个项:用户可以从菜单或工具栏按钮的“对齐”菜单中调用具体的对齐动作,如“顶端对齐”;或者点击“对齐”工具栏按钮,这样将使用上一次的对齐选项(如果是程序刚刚启动第一次使用对齐,则使用默认的对齐选项)。

为了让大家有些概念,下面是从主窗口部件的 `createMenusAndToolBars()` 函数中引用的关于设置对齐动作的代码:

```
QMenu *alignmentMenu = new QMenu(tr("Align"), this);
foreach (QAction *action, QList<QAction*>())
    << editAlignLeftAction << editAlignRightAction
    << editAlignTopAction << editAlignBottomAction
    alignmentMenu->addAction(action);
editAlignmentAction->setMenu(alignmentMenu);
```

接下来“编辑对齐”动作被加入到编辑菜单和编辑工具栏。用户使用这个菜单的时候只能选中具体的对齐动作其中之一,但如果使用工具栏的话就既可以点击工具栏按钮(即调用编辑对齐动作)也可以点击工具栏的菜单,这样可以选择具体的对齐动作之一。考虑到这种情况,代码中还需要考虑既有可能是调用了某个具体的对齐动作,也可能是调用了编辑对齐动作本身。还要注意, `updateUi()` 槽仅在有两个及更多选中项时才会使能对齐动作,因为对齐只有一个选中项或没有选中项的情况是没有意义的。

为了讲解方便,我们把这个槽分成三部分介绍。

```
void MainWindow::editAlign()
{
    QAction *action = qobject_cast<QAction*>(sender());
    if (!action)
        return;

    Qt::Alignment alignment = static_cast<Qt::Alignment>(
        action->data().toInt());
    if (action != editAlignmentAction) {
        editAlignmentAction->setData(action->data());
        editAlignmentAction->setIcon(action->icon());
    }
}
```

首先确定哪个动作调用了该槽,这就要求在动作的数据里保存对齐方式信息,这样在后面可以获取该信息。如果动作不是“编辑对齐”本身,则将编辑对齐动作的数据设置为选中的具体的对齐方式,并将图标设置为对齐方式对应的图标。这是为了保证当“编辑对齐”动作本身而不是具体的对齐方式被调用的时候,它会使用上一次设置过的对齐方式。

```
QList<QGraphicsItem*> items = scene->selectedItems();
QVector<double> coordinates;
populateCoordinates(alignment, &coordinates, items);
double offset;
if (alignment == Qt::AlignLeft || alignment == Qt::AlignTop)
    offset = *std::min_element(coordinates.constBegin(),
                                coordinates.constEnd());
else
    offset = *std::max_element(coordinates.constBegin(),
                                coordinates.constEnd());
```

用来对齐的算法非常简单,我们创建一个矢量(vector),保存需要的对齐方式所对应的所有项的边

界坐标,例如,如果向左对齐就保存 x 坐标,底端对齐就保存 y 坐标加上项的高度。然后计算位移(对左端或顶端对齐来说,是最小坐标;对右端和底端对齐来说是最大坐标),并按照位移和项的实际坐标的差值移动每个项。

`std::min_element()` 和 `std::max_element()` 函数是由 STL(标准模板库,即 Standard Template Library) 中的头文件 `<algorithm>` 提供的。这两个函数接受一个开始和结束迭代器(iterator),并返回迭代器所指序列中指向最小值(或最大值)元素的迭代器。所以,这里我们立即使用 `operator *()` 操作符把数值从迭代器中提取出来(如果想避免使用 STL,那就需要写自己的用一个序列做参数的 `min()` 和 `max()` 函数模板了,或者可以用 `qSort(coordinates); offset = coordinates.first();` 来获取最小值,使用 `coordinates.last()` 来获取最大值)。

```
if (alignment == Qt::AlignLeft || alignment == Qt::AlignRight) {
    for (int i = 0; i < items.count(); ++i)
        items.at(i)->moveBy(offset - coordinates.at(i), 0);
}
else {
    for (int i = 0; i < items.count(); ++i)
        items.at(i)->moveBy(0, offset - coordinates.at(i));
}
setDirty(true);
}
```

在这里我们遍历每一项,在水平方向或垂直方向上移动必要的距离以向左对齐(或向右对齐等)这些项。当所有的项都移动完毕后,则调用 `setDirty()`,以表明场景有明显的变化发生。

从用户的角度来看,当调用对齐动作时,选中项立刻就粘贴到了新的位置。在第 13 章中我们会再回过头来看一下这个方法,看看怎样使得对齐动作的过程动画化,以便对于用户来说显得比较明显些,同时使得移动动作在视觉上也更流畅些。

```
void MainWindow::populateCoordinates(const Qt::Alignment &alignment,
    QVector<double> *coordinates,
    const QList<QGraphicsItem> &items)
{
    QListIterator<QGraphicsItem> i(items);
    while (i.hasNext()) {
        QRectF rect = i.next()->sceneBoundingRect();
        switch (alignment) {
            case Qt::AlignLeft:
                coordinates->append(rect.x()); break;
            case Qt::AlignRight:
                coordinates->append(rect.x() + rect.width()); break;
            case Qt::AlignTop:
                coordinates->append(rect.y()); break;
            case Qt::AlignBottom:
                coordinates->append(rect.y() + rect.height()); break;
        }
    }
}
```

这个函数遍历给定的项的列表,将 `double` 类型的 x 或 y 坐标填充到矢量(vector)中。`QGraphicsItem::sceneBoundingRect()` 函数返回项在场景坐标系中的边界矩形。更常见的用法是用 `QGraphicsItem::boundingRect()` 返回项在它的自身坐标系的边界矩形,但这里我们计划在场景中移动项(如排列项),所以需要场景坐标系中的坐标。从 `QGraphicsItem` 的 API 中选出的重要函数和 API 使用到的关键枚举值已在前面的表 11.1 至表 11.7 列出。

绘图程序另一个常见的需求是水平或垂直排布项的功能。意即给定三个及更多的项,保持两个端点的项不动,把中间的项按照相同的间隔排布在端点项之间。实现这个功能无法教给大家任

何前面已经介绍过的、关于图形/视图架构的内容之外的东西,因此,为 Page Designer 添加这个功能就作为练习留给读者。

在结束这一操作图形项的小节之前,我们看一下几个槽。

```
void MainWindow::selectionChanged()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    if (items.count() == 1) {
        if (QObject *item = dynamic_cast<QObject*>(items.at(0))) {
            if (item->property("brush").isValid())
                brushWidget->setBrush(
                    item->property("brush").value<QBrush>());
            if (item->property("pen").isValid())
                penWidget->setPen(
                    item->property("pen").value<QPen>());
            if (item->property("angle").isValid()) {
                transformWidget->setAngle(
                    item->property("angle").toDouble());
                transformWidget->setShear(
                    item->property("shearHorizontal").toDouble(),
                    item->property("shearVertical").toDouble());
            }
        }
    }
    updateUi();
}
```

这个槽连接到了 `QGraphicsScene::selectionChanged()` 信号,我们用它确保工具箱窗口部件正确显示出选中项的属性[如前面所述,使用 `dynamic_cast <>()` 让程序不依赖 RTTI (Run Time Type Information 运行时类型信息)是否可用,当然还有其他若干方法可以避免这个问题]。

这里我们关心选中项发生的改变,比如变成只有一项被选中的情况。如果有两个及更多的项被选中,我们不知道该显示哪个画刷、画笔或变换的属性信息,所以这种情况下保持工具箱不变。当多项被选中时不能禁用工具箱,因为用户可能想要一次性应用针对多项的更改,比如旋转选中的项或者改变它们的画刷。

我们再次使用了 Qt 的元对象系统,以使函数尽量通用,这样即使添加了新的自定义项类型也不需要修改函数(正如前面注明的,如果不想让自定义项类型从 `QObject` 派生,可以使用 `qgraphic-item_cast <>()` 函数替代)。在前面介绍的 `connectItem()` 函数中,我们使用了项的(类的)元对象和 `QMetaObject::indexOfProperty()` 函数查看特定的项是否有某个属性。这里用一种更直接(但更含糊)的方式,使用 `QObject::property()` 方法。这个方法接收一个属性名,返回 `QVariant`,如果对象没有给定名称的属性,就返回一个无效的 `QVariant`^①。

选中的项数(无选中项、单个项选中、两个或更多选中项)对用户界面有深远的影响。如果无选中项那么工具箱部件应该被禁用,因为工具箱没有任何作用;如果正好一个项被选中,“编辑选中项…”动作应被使能;如果两个或更多项被选中,那么对齐相关的动作应该被使能。所有这些以及更多其他逻辑由 `updateUi()` 槽处理,我们很快会介绍这个函数。

如 Page Designer 应用程序所示,如果项是一个方框或表情符号, `BrushWidget` 和 `PenWidget` 将显示它的画刷和画笔, `TransformWidget` 将显示项的旋转和修剪 (shear) 属性(将在后面介绍 `BrushWidget`)。回想前面介绍的内容,如果用户在工具箱窗口部件里修改了属性,如修改画刷,将通知

① 如果不存在指定名称的属性,或者属性本身是无效的 `QVariant`, 那么 `QObject::property()` 函数返回一个无效的 `QVariant`, 所以如果想确切知道一个对象是不是有某个特定的属性,必须用 `QMetaObject::indexOfProperty()` 方法。

每个有画刷属性的项;但仅仅被选中的项会响应并更新画刷,在后面涉及到自定义图形项时会介绍。

12.1.3.4 显示和隐藏参考网格

用 `QGraphicsScene::setBackgroundBrush()` 设置场景的背景为一个合适的画刷,或者通过重新实现 `QGraphicsScene::drawBackground()` 函数大概是显示参考网格最简单也是最好的方法了。我们并没有使用两者的任一种,这仅仅是为了展示如何使用 `QGraphicsItemGroup` 以及如何有选择性地保存或操作项。

在页面设计器程序中我们有一个 `viewShowGridAction` 切换动作 (toggle action), 连接到 `viewShowGrid()` 槽, 该函数创建并显示/隐藏参考网格。

```
void MainWindow::viewShowGrid(bool on)
{
    if (!gridGroup) {
        const int GridSize = 40;
        QPen pen(QColor(175, 175, 175, 127));
        gridGroup = new QGraphicsItemGroup;
        const int MaxX = static_cast<int>(std::ceil(scene->width())
            / GridSize) * GridSize;
        const int MaxY = static_cast<int>(std::ceil(scene->height())
            / GridSize) * GridSize;
        for (int x = 0; x <= MaxX; x += GridSize) {
            QGraphicsLineItem *item = new QGraphicsLineItem(x, 0, x,
                MaxY);
            item->setPen(pen);
            item->setZValue(std::numeric_limits<int>::min());
            gridGroup->addToGroup(item);
        }
        ...
        scene->addItem(gridGroup);
    }
    gridGroup->setVisible(on);
}
```

这个函数被调用时,如果参考网格组不存在,就创建它。例如,可能程序刚刚启动,或刚刚清空了场景。我们设置绘制网格的画笔为半透明的浅灰色。由于没有指定画笔的宽度,它将默认为宽度 0,表示 1 像素宽的“装饰画笔”(cosmetic pen)。“装饰画笔”(不管宽度为何)总是按照指定的宽度绘制,不管实行的是何种变换(除了宽度为 0 的情况将按宽度为 1 处理)。非装饰画笔与之相比,它的宽度将按现行的缩放比例缩放。

我们创建一个新的 `QGraphicsItemGroup` 并计算场景中最远的 x 和 y 坐标。然后 x 坐标从 0 到 `MaxX` 循环遍历所有的 `GridSize` 增量,每次循环创建一个标准的 `QGraphicsLineItem`,并把 (x_1, y_1, x_2, y_2) 坐标传入。由于我们传入两个完全相同的 x 坐标而 y 坐标不同,就得到了垂直的一列。行创建完毕后把画笔设回原先的值,然后把该项的 z 值设置为一个比较大的负值,这样可以保证它位于其他项之下,最后把它添加到网格组中。绘制水平线用了类似的循环,由于代码结构与列出的代码相同,这部分代码省略。

`QGraphicsItem` 创建时所有的图形项标志为禁用状态,因此默认情况下图形项不能被用户移动或者选中。这正是我们希望网格呈现的行为。

所有线条的项创建完毕并加入到组中,那么就把该组添加到场景。最后按照相应动作的 `toggle(bool)` 信号传给槽的布尔值参数设置组(即包括了它包含的所有线条项)为显示或隐藏。

顺便提一句, `std::ceil()` 函数(`<cmath>` 提供)返回不小于其参数的最小整数, `std::numeric_limits<int>::min()` 函数(`<limits>` 提供)返回最小整数[如最负值(most negative)]。

这里还没有介绍的一个函数是 `editClearTransforms()`，我们将在后面“图形项变换”小节的末尾介绍。

12.1.3.5 保持更新用户界面

把程序状态显示到用户界面一个更省事的方法是根本就不显示该状态。用这种方法的话，用户可以在任何时候调用任何动作，由每个动作判断是否有意义，如果无意义就不做任何事。例如可以把“粘贴”动作设为总是使能，即使没有可粘贴的东西，类似的即使没有选中的项也把“复制”动作使能（这种情况下无可复制的内容）。

可惜把所有动作使能会把用户搞糊涂，比如他们可能调用了“粘贴”动作，然后不明白为何什么都没有显示出来。所以在可能的范围之内还是应该根据程序的状态来使能或禁用动作。有时候这可能不容易实现。比如当用户选择了一个方框项、一个表情符号和一个文本项，我们该使能哪些工具箱窗口部件呢？只能使能对所有项均适用的工具箱，针对这种情况就只能使能变换工具箱了。Page Designer 程序中为了用户使用方便，我们决定使能可以适用于至少一个选中项的工具箱，所以针对这个例子可以使能变换、画笔和画刷工具箱，尽管文本项没有画笔或画刷属性。这个办法能很好地工作，因为当修改比如画刷时，文本项会忽略该修改，而选中的方框和表情符号项则会正确地应用修改。

Page Designer 中每当有明显的更改发生时就会调用 `setDirty()` 函数。调用“保存”动作时该函数被调用并传入 `false` 作为参数，因为保存之后场景就不再是“脏”（dirty，表示有未保存的修改）的状态了。但对大多数其他改变来说，这个函数得到的参数为 `true`（参数的默认值）。

```
void MainWindow::setDirty(bool on)
{
    setWindowModified(on);
    updateUi();
}
```

我们利用了主窗口的 `windowModified` 属性，这样就不用维护自己的布尔型的表示有未保存修改的变量。当“脏”状态发生改变时调用 `updateUi()`（在别的地方也调用了这个函数——比如在 `editCopy()` 和 `selectionChanged()` 函数中），以正确使能或禁用应用程序的动作（actions）和工具箱部件。

```
void MainWindow::updateUi()
{
    fileSaveAction->setEnabled(isWindowModified());
    bool hasItems = sceneHasItems();
    fileSaveAsAction->setEnabled(hasItems);
    fileExportAction->setEnabled(hasItems);
    filePrintAction->setEnabled(hasItems);
    int selected = scene->selectedItems().count();
    editSelectedItemAction->setEnabled(selected == 1);
    editCopyAction->setEnabled(selected >= 1);
    editCutAction->setEnabled(selected >= 1);
    QClipboard *clipboard = QApplication::clipboard();
    const QMimeData *mimeType = clipboard->mimeType();
    editPasteAction->setEnabled(mimeType &&
        (mimeType->hasFormat(MimeType) || mimeType->hasHtml() ||
         mimeType->hasText()));
    editAlignmentAction->setEnabled(selected >= 2);
    editClearTransformsAction->setEnabled(selected >= 1);
    transformWidget->setEnabled(selected >= 1);
    bool hasBrushProperty;
    bool hasPenProperty;
    getSelectionProperties(&hasBrushProperty, &hasPenProperty);
    brushWidget->setEnabled(hasBrushProperty);
    penWidget->setEnabled(hasPenProperty);
}
```



当有未保存的修改时,“保存”动作使能。当场景中至少有一个项时,“另存为”、“导出”和“打印”动作被使能。当刚好仅有一个项被选中时,“编辑选中的项…”动作被使能;至少有一项被选中时,“复制”和“剪切”动作使能。当系统剪贴板中有 Page Designer 的 MIME 格式的数据、HTML 或纯文本格式的数据时,“粘贴”动作被使能;我们在前面介绍了剪贴板的处理。当至少两个项被选中时,“对齐”动作(包括“左端对齐”、“右端对齐”及其他动作)被使能。至少一项选中时,“清除变换”动作被使能。严格来讲应该在选中项(或至少选中的项之一)有非零值的旋转或修剪时才使能“清除变换”动作,检查这个属性值不难,但我们把这个改进作为练习留给大家。

至少有一项被选中时我们使能变换工具箱部件,因为所有 Page Designer 程序中的自定义项都支持变换。但仅当选项(或至少选中项之一)有画刷或画笔属性时使能画刷、画笔工具箱部件。

为完整起见,我们将概览 `updateUi()` 槽的两个帮助函数。

```
bool MainWindow::sceneHasItems() const
{
    foreach (QGraphicsItem *item, scene->items())
        if (item != gridGroup && item->group() != gridGroup)
            return true;
    return false;
}
```

当场景中至少有一个项(不包括参考网格)时这个函数返回 `true`。

```
void MainWindow::getSelectionProperties(bool *hasBrushProperty,
                                       bool *hasPenProperty) const
{
    Q_ASSERT(hasBrushProperty && hasPenProperty);
    *hasBrushProperty = false;
    *hasPenProperty = false;
    foreach (QGraphicsItem *item, scene->selectedItems()) {
        if (QObject *object = dynamic_cast<QObject*>(item)) {
            const QMetaObject *metaObject = object->metaObject();
            if (metaObject->indexOfProperty("brush") > -1)
                *hasBrushProperty = true;
            if (metaObject->indexOfProperty("pen") > -1)
                *hasPenProperty = true;
            if (*hasBrushProperty && *hasPenProperty)
                break;
        }
    }
}
```

该函数遍历每个选中的项,检查它是否有画刷或画笔属性。这里引入了一个提升效率的小技巧,如果画刷和画笔属性都为 `true`,那么就可以直接返回(在前面提醒注意的,使用 `dynamic_cast <>()` 使程序不依赖 RTTI 是否可用,但可以用其他方法绕过这个问题)。

12.2 增强 QGraphicsView 的功能

QGraphicsView 类不直接支持缩放功能,所以我们创建了一个微型的子类(所有的实现都放在一个头文件中)提供必要的功能,同时借此机会打开反走样和提供橡皮筋(rubber band)选择功能。下面是 GraphicsView 类完整的定义:

```
class GraphicsView : public QGraphicsView
{
    Q_OBJECT

public:
    explicit GraphicsView(QWidget *parent=0) : QGraphicsView(parent)
    {
        setDragMode(RubberBandDrag);
    }
}
```

```

        setRenderHints(QPainter::Antialiasing|
                       QPainter::TextAntialiasing);
    }

public slots:
    void zoomIn() { scaleBy(1.1); }
    void zoomOut() { scaleBy(1.0 / 1.1); }

protected:
    void wheelEvent(QWheelEvent *event)
    { scaleBy(std::pow(4.0 / 3.0, (-event->delta() / 240.0))); }

private:
    void scaleBy(double factor) { scale(factor, factor); }
};

```

构造函数的开头打开橡皮筋拖曳模式,表示如果用户点击并拖曳,将拉开一个橡皮筋(比如在某些平台上为一个填充了半透明颜色的矩形),每个在橡皮筋矩形中或接触到矩形的项被选中。默认的拖曳模式是 `QGraphicsView::NoDrag`,即什么都不做。另外一个支持的拖曳模式为 `QGraphicsView::ScrollHandDrag`,设置为这个模式时点击和拖曳将滚动整个视图。

我们提供了两个缩放视图的槽函数以支持缩放功能,另外还重新实现了鼠标滚轮事件处理函数,按照鼠标滚轮滚动的量来缩放——向前滚动为缩小,向后滚动为放大。`QWheelEvent::delta()` 返回的值代表用“步数”表示的滚轮滚动的距离(通常一步对应滑轮滚动 15°),正值代表向前滚动,负值表示向后滚动。`std::pow()` 函数(`<cmath>` 头文件提供)计算以第二个参数为指数、第一个参数为基数的值。本质上我们要做的是把每个鼠标滚动的“步”乘以 $1\frac{1}{3}$ 作为放大或缩小的倍数。

缩放工作在私有方法 `scaleBy()` 中完成,这个函数简单地调用 `QGraphicsView::scale()` 方法,在水平方向和垂直方向使用单一的缩放因子缩放,以保持视图的宽高比。

默认情况下, `QGraphicsView` 使用滚轮事件实现滚屏操作,通过重新实现 `wheelEvent()` 事件处理函数,可以有效地去掉这个功能。也就是说,用户只能在鼠标滚动条之上时才能用鼠标滚轮控制滚屏。

12.3 创建可停靠的工具箱窗口部件

Page Designer 应用程序的主窗口有三个可停靠的部件——变换(旋转和修剪)、画刷和画笔。三个部件在结构上相似,逻辑上都用于同样的两个目的:显示单个选中部件的相关属性(例如显示它的画刷颜色和风格)和当用户操作可停靠部件的编辑部件时,更改选中的一个或多个项的属性。

在结构和功能上,所有 Page Designer 的停靠窗口部件都是类似的,所以只介绍其中之一——`BrushWidget`。其他部件的代码当然包含在程序的源码中,在图 12.1 中可以看到所有三个部件。

按照常规,先介绍类在头文件中的定义,但省略私有槽和私有成员函数,在浏览公有槽和成员函数时再根据需要介绍。

```

class BrushWidget : public QWidget
{
    Q_OBJECT

public:
    explicit BrushWidget(QWidget *parent = 0);
    QBrush brush() const { return m_brush; }

public slots:
    void setBrush(const QBrush &brush);

```



```
signals:
    void brushChanged(const QBrush &brush);
    ...
};
```

这个部件有一项私有成员数据——QBrush 类型的 `m_brush`, 该类为这个变量提供了合适的获取函数 (getter) 和设定 (setter) 函数, 这样画刷可以通过激活一个信号/槽的连接来设定。如果画刷发生了改变 (例如用户修改了颜色或者风格) `brushChanged()` 信号将被发出。我们在前面看到每个有画刷属性的项 (即方框和表情符号项) 都连接到这个信号 (在后面介绍自定义项的实现时将介绍项是如何响应信号的)。

```
BrushWidget::BrushWidget(QWidget *parent)
    : QWidget(parent)
{
    createWidgets();
    setBrush(QBrush());
    createLayout();
    createConnections();
    setFixedSize(minimumSizeHint());
}
```

构造函数把自己的大部分工作转移到私有帮助函数进行。它设置一个风格为 `Qt::NoBrush` (所以该画刷实际上没有任何作用) 的黑色画刷, 当子窗口部件创建好并分布开来后, 设置画刷窗口部件的最小尺寸提示 (hint) 为固定值, 因为让用户调整它的大小没有多大意义 (如果有兴趣可以尝试注释掉三个工具箱部件的 `setFixedSize()` 调用, 看看是什么效果)。

```
void BrushWidget::createWidgets()
{
    colorComboBox = new QComboBox;
    foreach (const QString &name, QColor::colorNames()) {
        QColor color(name);
        colorComboBox->addItem(colorSwatch(color), name, color);
    }

    styleComboBox = new QComboBox;
    typedef QPair<QString, Qt::BrushStyle> BrushPair;
    foreach (const BrushPair &pair, QList<BrushPair>())
        << qMakePair(tr("No Brush"), Qt::NoBrush)
        << qMakePair(tr("Solid"), Qt::SolidPattern)
        ...
        << qMakePair(tr("Diagonal Cross"), Qt::DiagCrossPattern))
    styleComboBox->addItem(brushSwatch(pair.second), pair.first,
        pair.second);
}
```

该窗口部件包含 4 个子部件——两个文本标签 (label) 和两个组合框 (combobox)。为了协助用户选择画刷的颜色或式样 (style), 我们提供样本 (小图片) 说明它的值, 而不仅仅列出名称。每种颜色用一个圆形的、填充为相关颜色的图片, 每种画刷风格用填充了当前颜色 (初始值为黑色)、使用对应画刷式样的方形图片。

针对每种颜色我们添加一个带有颜色样本的组合框项、颜色的名称 (使用 HTML 颜色语法的字符串, 如红色为 `"#FF0000"`) 以及一个保存颜色本身的 `QVariant` 数据。静态成员函数 `QColor::colorNames()` 返回一个经过排序的、人类可读 (human-readable) 的颜色名称 [比如 `"palegreen"` (淡绿色)、`"red"` (红色), 以此类推] 组成的 `QStringList`^①。对画刷式样风格采用相同的方法, 赋予每个组合框项一个样本, 说明画刷的式样、式样的名称和保存式样的枚举值的 `QVariant` 数据——当然在上面展示的代码中, 我们省略了大多数画刷对 (颜色和式样), 因为它们都遵循同样的模式。

① 本书撰写之际, 文档并未明确说明颜色名称是排序的, 保守起见最好是获取名称后, 在添加项之前调用 `QStringList::sort()`, 或者在填充到组合框之后调用 `QComboBox::model() -> sort(0)` 按第一列递升排序。

正如我们在第9章中注明的,在Qt的foreach语句(construct)中出现的第一个逗号用于从序列中分离单个项,所以我们不能用包含逗号项。在这里用惯常的创建typedef的方法解决了这个问题。

另外一个需要注意的方面是仅创建了组合框,而没有创建文本标签。我们将让布局类创建文本标签,如很快将要看到的那样。

这个程序有5个样本函数:colorSwatch()、brushSwatch()、penStyleSwatch()、penCapSwatch()和penJoinSwatch()。我们将观察其中最简单、作为代表的brushSwatch()函数[其他函数更复杂,但它们的复杂之处并不在于我们感兴趣的方面——例如颜色样本(color swatch)可以添加一个对比色的“C”字符,画笔连接样本(pen join swatch)绘制两条相交的线段组成的折线以显示出连接的式样]。

```
QPixmap brushSwatch(const Qt::BrushStyle style, const QColor &color,
                    const QSize &size)
{
    QString key = QString("BRUSHSTYLESWATCH:%1:%2:%3x%4")
        .arg(static_cast<int>(style)).arg(color.name())
        .arg(size.width()).arg(size.height());
    QPixmap pixmap(size);
    if (!QPixmapCache::find(key, &pixmap)) {
        pixmap.fill(Qt::transparent);
        QPainter painter(&pixmap);
        painter.setRenderHint(QPainter::Antialiasing);
        painter.setPen(Qt::NoPen);
        painter.setBrush(QBrush(color, style));
        painter.drawRect(0, 0, size.width(), size.height());
        painter.end();
        QPixmapCache::insert(key, pixmap);
    }
    return pixmap;
}
```

该函数接收一个画刷式样、一个可选的颜色(默认值为黑色)以及一个可选的尺寸(默认大小为24×24像素)为参数,然后返回一个按照指定式样、颜色、大小的画刷绘制的正方形图片。

我们不在每次函数被调用时创建一个新图片,而是将图片缓存下来——默认可以使用高达10兆字节的空間,并且可以通过调用QPixmapCache::setCacheLimit()函数来修改。要想使用缓存,必须用唯一的键值标志每个需要缓存的图片,这里用一个标志图片为画刷样本(相对于颜色样本、画笔式样样本及其他)的字符串,紧接着是标志样本的信息——本例中是画刷的式样(整数表示)、颜色(HTML颜色字符串表示)和图片的宽和高。例如,纯棕色24×24的画刷的键值为"BRUSHSTYLESWATCH:1:#a52a2a:24×24"。

创建好键值之后创建正确大小的图片。QPixmapCache::find()函数用于用给定的键值从缓存中获取一个图片。该函数如果找到键值,就返回true并填充传入的QPixmap指针,或者Qt 4.5及更早的版本传入的非常量引用(如no&);否则返回false。所以第一次请求某图片时键值没有找到,那么我们就自己填充图片。先用透明色填充图片,然后创建一个绘制对象(QPainter),用指定的画刷绘制一个占满整个图片的矩形。虽然我们会绘制整个图片,但由于多数画刷式样不是纯色的,会让一些背景透过,所以在最开始还是必须用透明色或者其他颜色填充图片。创建完毕之后根据键值把图片添加到缓存,在函数末尾返回获取或创建的图片。

所有其他的样本函数与这个函数在结构上都是相同的——不同之处仅在于它们使用不同的键值,且在图片上绘制不同的内容。注意使用的键值不能以字符串"\$qt"打开,因为Qt自身使用QPixmapCache时,用这个字符串作为所有自己的键值的前缀。

```
void BrushWidget::createLayout()
{
    QFormLayout *layout = new QFormLayout;
    layout->addRow(tr("Color"), colorComboBox);
    layout->addRow(tr("Style"), styleComboBox);
    setLayout(layout);
}
```

我们选择展示 `createLayout()` 函数,因为它使用了 Qt 4.4 引入的 `QFormLayout` 类,还为我们创建了两个文本标签。用这种方式创建文本标签还有一个精妙的功能,那就是 Qt 自动把文本标签设置为添加部件的伙伴(buddy),意味着如果键盘焦点切换到文本标签时,会立刻传递到它的伙伴。

```
void BrushWidget::createConnections()
{
    connect(colorComboBox, SIGNAL(currentIndexChanged(int)),
            this, SLOT(updateColor(int)));
    connect(styleComboBox, SIGNAL(currentIndexChanged(int)),
            this, SLOT(updateStyle(int)));
}
```

这个函数为了完整性而展示。如果用户修改了颜色或者画刷的式样,对应的槽被调用,所以,所有连接的部件都会接到通知,这样颜色改变时可以用新设定的颜色更新画刷样本。

我们完成了构造函数及其私有帮助函数的介绍,下一步将介绍公有的 `setBrush()` 槽,然后是私有槽和私有的帮助方法。

```
void BrushWidget::setBrush(const QBrush &brush)
{
    if (m_brush != brush) {
        m_brush = brush;
        colorComboBox->setCurrentIndex(
            colorComboBox->findData(m_brush.color()));
        styleComboBox->setCurrentIndex(
            styleComboBox->findData(
                static_cast<int>(m_brush.style())));
    }
}
```

当视图中刚好仅有一个选中项时这个槽被调用——在项有画刷属性的条件下,例如方框和表情符号项。作为响应,该槽设置私有画刷为传入的值,设置两个组合框的当前项,使之匹配新的画刷颜色和式样。

`QComboBox::findData()` 方法接收 `QVariant` 参数,返回第一个匹配项的索引(或 -1)。我们可以直接使用 `QColor`,但对于枚举值必须把它强制转换为整数,因为这是枚举值在 `QVariant` 中存储的方式。

```
void BrushWidget::updateColor(int index)
{
    m_brush.setColor(colorComboBox->itemData(index).value<QColor>());
    updateSwatches();
    emit brushChanged(m_brush);
}
```

不论何时画刷的颜色发生改变(可能是用户操作 `colorComboBox` 的结果,或者源于一个项被选中并调用了 `setBrush()` 槽)这个函数都会被调用。

`QComboBox::itemData()` 函数返回与给定的索引关联的项的 `QVariant` 数据(默认是无效的 `QVariant`)。另外由于 `QVariant` 仅提供 QtCore 类型的转换函数(`QVariant::toInt()`、`QVariant::toSize()`及其他),对于其他类型必须调用 `QVariant::value<T>()` 模板函数,指定想要返回的值的类型。

使用新颜色更新私有画刷后,我们调用私有方法 `updateSwatches()` 以确保画刷的式样样品按照新颜色显示。在最后发出 `brushChanged()` 信号,把新画刷通知给连接的对象。

```
void BrushWidget::updateStyle(int index)
{
    m_brush.setStyle(static_cast<Qt::BrushStyle>(
        styleComboBox->itemData(index).toInt()));
    emit brushChanged(m_brush);
}
```

该方法是针对画刷式样的 `updateColor()` 函数的等价体。画刷式样用一个枚举值指定,所以先把给定项的 `QVariant` 数据作为一个整数读取,然后在更新私有画刷之前把它转换为合适的枚举类型。在最后发出带有新画刷的 `brushChanged()` 信号。

在这个函数中不需要调用 `updateSwatches()`,因为任何时候一旦颜色发生变化,所有的组合框中的样本都会刷新,所以当用户操作画刷样式组合框时,或者用编程方式修改了组合框的索引时,它的样本已经使用了正确的颜色。

```
void BrushWidget::updateSwatches()
{
    QColor color = colorComboBox->itemData(
        colorComboBox->currentIndex()).value<QColor>();
    for (int i = 0; i < styleComboBox->count(); ++i)
        styleComboBox->setItemIcon(i, brushSwatch(
            static_cast<Qt::BrushStyle>(
                styleComboBox->itemData(i).toInt()), color));
}
```

为了让用户尽可能容易地看到修改画刷颜色的效果,每当画刷颜色改变时更新 `styleComboBox` 中的画刷式样样本,让它们使用相同的颜色。其效果很容易看到,通过运行 `Page Designer` 应用程序,点击画刷式样组合框,选择 `NoBrush` 之外的任一画刷式样,然后点击画刷颜色组合框,用上下方向键更改颜色——同时观察画刷样本组合框中的样本随着选择的颜色变化(在一些系统平台中必须在第一次点击颜色组合框后按下 `Esc` 键,才能使组合框在按下方向键时保持可见)。通过在画笔颜色组合框上使用上下键并观察画笔式样、笔端式样(`cap`)、笔划连接处的式样(`join`)样本颜色的改变,可以达到同样的效果。

该函数先获取当前的颜色,然后遍历画刷式样组合框中的每一项,修改它使用的图标(隐式地调用了接收 `QPixmap` 参数的 `QIcon` 构造函数)为调用 `brushSwatch()` 函数返回的图片。我们给 `brushSwatch()` 函数的第一个参数传入画刷的式样,该式样通过获取 `QVariant` 的整型数据并转换为合适的枚举类型得到。我们还传入颜色组合框的当前颜色作为第二个参数,而不使用默认的黑色。

`TransformWidget` 和 `PenWidget`(在以均未做介绍)在结构上和逻辑上与前面介绍的 `BrushWidget` 非常相似。

截至目前已经介绍了 `Page Designer` 应用程序所有的基础结构,包括如何保存、载入和导出场景,如何添加和操作单个/多个项,以及如何操作一组选中的项。只剩下一个需要介绍的,即“自定义图形项”本身,这正是下一节的主题。

12.4 创建自定义图形项

所有的图形/视图项都以 `QGraphicsItem` 作为直接基类或间接基类。大多数 `QGraphicsItem` 的便利子类已经与图形/视图架构一道在 Qt 4.2 引入, `QGraphicsProxyWidget` 和 `QGraphicsWidget` 在 Qt 4.4 时加入, `QGraphicsObject` 和 `QGraphicsWebView` 在 Qt 4.6 时加入[Qt 4.7 有可能会增加用于

在场景中显示视频(使用 Qt 的底层 QtMultimedia 模块)的 QGraphicsVideoItem 类]。类的层次结构在前面介绍了,一部分方法和枚举值也做了介绍(在表 11.1 和表 11.7 中)。

已经有了这么多图形项的类,这就意味着,如果要绘制自定义的形状,大多数情况下没有必要创建子类。归根结底,所有形状都可以用 QGraphicsEllipseItem、QGraphicsLineItem、QGraphicsPathItem、QGraphicsPolygonItem 和 QGraphicsRectItem 绘制,所以子类化图形项的主要目的是为了提供自定义的行为。

本节我们将介绍三种不同类型的自定义图形项,以展示各种不同的方案。介绍的第一个子类是一个 QGraphicsTextItem 的子类,它仅增加了一些简单的行为,所有的绘制留给 Qt 完成。第二个是 QObject 和 QGraphicsRectItem 的子类,增加了更多复杂的行为,包括键盘按下与鼠标交互。第三个是 QGraphicsObject 子类(对 Qt 4.5 及更老版本为 QObject 和 QGraphicsItem 的子类),实现了定制行为并绘制自身,还提供了 boundingRect() 和 shape() 的实现。事实上最后一个类如果是 QGraphicsPathItem 的子类的话,就可以不需要自己绘制了(对于大多数形状皆是如此),但这里我们想要展示一个同时实现行为和外观的例子。

为方便起见,在单独的头文件中定义了项的类型值:

```
const int BoxItemType = QGraphicsItem::UserType + 1;
const int SmileyItemType = QGraphicsItem::UserType + 2;
const int TextItemType = QGraphicsItem::UserType + 3;
```

希望可以帮助我们防止误把同一个类型值赋予两个自定义项。

12.4.1 增强 QGraphicsTextItem

我们用 QGraphicsTextItem 类(继承了 QObject 和 QGraphicsItem)在场景中显示 Qt 的富文本(rich text)(对于纯文本,即文本内容用单一的字体和颜色,可以用 QGraphicsSimpleTextItem 替代)。

为了提供定制的行为,我们要从 QGraphicsTextItem 派生,特别是当想让用户可以在界面上进行项的旋转和修剪,而且可以编辑显示出来的文本时。另外还想让该类通过自定义的 dirty() 信号发布修改消息,并能从 QDataStream 读出、写入项本身。绘制工作以及边界矩形和形状的计算留给基类完成。

首先看一看自定义 TextItem 类在头文件中定义,但省略掉了私有部分。

```
class TextItem : public QGraphicsTextItem
{
    Q_OBJECT
    Q_PROPERTY(double angle READ angle WRITE setAngle)
    Q_PROPERTY(double shearHorizontal READ shearHorizontal
                WRITE setShearHorizontal)
    Q_PROPERTY(double shearVertical READ shearVertical
                WRITE setShearVertical)

public:
    enum {Type = TextItemType};

    explicit TextItem(const QPoint &position, QGraphicsScene *scene);
    int type() const { return Type; }

    double angle() const { return m_angle; }
    double shearHorizontal() const { return m_shearHorizontal; }
    double shearVertical() const { return m_shearVertical; }

public slots:
    void setAngle(double angle);
```



```

void setShearHorizontal(double shearHorizontal)
{ setShear(shearHorizontal, m_shearVertical); }
void setShearVertical(double shearVertical)
{ setShear(m_shearHorizontal, shearVertical); }
void setShear(double shearHorizontal, double shearVertical);
void edit();

signals:
    void dirty();

protected:
    QVariant itemChange(GraphicsItemChange change,
        const QVariant &value);
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent*) { edit(); }
    ...
};

```

如上一章所述,提供一个 Type 枚举类型,并重新实现 type() 函数(以支持用于转换 QGraphicsItem 指针为正确的 QGraphicsItem 子类的 qgraphicsitem_cast <>() 函数)是个很好的做法。

我们把项的旋转和修剪做成了属性,这样就可以通过 Qt 的属性系统查询和设定它们的值了——例如,在 selectionChanged() 方法中查询,在 editClearTransforms() 函数中设定。另外把这两个属性定义为 double 类型而不用 qreal,这是为了保证数据存储在载入的正确性(在第 3 章说明了不能把 qreal 与 QDataStream 一起使用)。

前两个槽让项可以响应 TransformWidget 工具箱中发生的更改。itemChange() 函数用于在更改有影响时发送 dirty() 信号(比如,如果项的位置或变换发生了改变),但在项被选中或取消选中时不发送。

在 Page Designer 程序中我们采用一个传统方法——提供 edit() 槽函数,这样就可以有提供基于项的编辑操作的“编辑选中项...”动作了(editSelectedItem() 槽在前面介绍过)。另外还重新实现了鼠标双击事件处理函数以提供另一种调用 edit() 的方式。

除了类的定义,头文件还包含了两个其他的声明。

```

QDataStream &operator<<(QDataStream &out, const TextItem &textItem);
QDataStream &operator>>(QDataStream &in, TextItem &textItem);

```

这两个操作符由 readItems() 和 writeItems() 方法使用,在前面介绍过。

下面看看在头文件中没有实现的所有方法,以及全局的流操作符,先从构造函数开始。

```

TextItem::TextItem(const QPoint &position, QGraphicsScene *scene)
: QGraphicsTextItem(), m_angle(0.0), m_shearHorizontal(0.0),
  m_shearVertical(0.0)
{
    setFont(QFont("Helvetica", 11));
    setFlags(QGraphicsItem::ItemIsSelectable|
        QGraphicsItem::ItemSendsGeometryChanges|
        QGraphicsItem::ItemIsMovable);
    setPos(position);
    scene->clearSelection();
    scene->addItem(this);
    setSelected(true);
}

```

TextItem 构造完毕后,先设定它的角度和修剪值为 0.0,然后设置一个初始位置(场景坐标)和它所属的场景。

我们希望用户可以选中和移动该项,所以设置合适的标志以允许这些操作。在 Qt 4.5 及以前的版本中,当项的几何信息(geometry,如项的大小)发生改变时,itemChanged() 函数被调用;但从 Qt 4.6 开始,为性能考虑不再调用该函数,除非像本例这样,通过设定 ItemSendsGeometryChanges 标志显式要求(在上一章的“Qt 4.6 图形/视图的行为变化的”阴影部分中介绍了这一点)。下一

步设置项的位置、清除已存在的选择并把项加到场景中。在函数的最后,我们选中该项,这样用户可以直接对它进行删除或编辑。

```
void TextItem::edit()
{
    QWidget *window = 0;
    QList<QGraphicsView*> views = scene()->views();
    if (!views.isEmpty())
        window = views.at(0)->window();
    TextItemDialog dialog(this, QPoint(), scene(), window);
    if (dialog.exec())
        emit dirty();
}
```

当单一的 `TextItem` 被选中,且用户(通过菜单或工具栏)调用“编辑选中的项...”动作或当用户双击一个 `TextItem` 时,调用该槽。

自定义 `TextItemDialog` (代码未列出,但图 12.3 显示了截图)是一个添加/编辑风格的对话框,提供给用户编辑文本的方法,包括使用不同的字体和颜色。我们希望确保对话框在合适的位置弹出,而且不会在任务栏增加额外的项,只要给它设置一个顶层窗口作为父窗口即可实现。可是我们在手边没有现成的顶层窗口指针。一个解决办法是调用 `QApplication::topLevelWidgets()`,然后使用列表中第一个非隐藏的窗口的指针,然而,虽然这样能得到一个顶层窗口,但未必是最合适的。所以我们采用另一个方法,获得与项的场景关联的视图列表(在本程序中只有一个)。然后调用 `QWidget::window()` 函数获得视图的顶层窗口(如果视图被用做独立的顶层窗口则为视图本身)。

`TextItemDialog` 在内部使用了在第 9 章介绍的 `TextEdit` 类,但没有用到文本对齐功能。对话框很智能(对话框有应用程序的信息并且基本上可以独立运作),如果用户点击 OK 按钮,它将更新 `TextItem` 并安排其重绘,这样只需要在对话框被接受时发出 `dirty()` 信号即可。

```
QVariant TextItem::itemChange(GraphicsItemChange change,
                              const QVariant &value)
{
    if (isDirtyChange(change))
        emit dirty();
    return QGraphicsTextItem::itemChange(change, value);
}
```

当项的状态发生改变时这个函数被 Qt 调用。但正如前面提醒注意的,从 Qt 4.6 开始,对于一些特定的改变(比如那些影响到项的位置和变换的改变),只有当通过设定正确的 `QGraphicsItem::GraphicsItemChange` 标志(即 `QGraphicsItem::ItemSendsGeometryChanges`)显式地请求时,Qt 才会调用这个函数(回想起即使设置了这个标志,也仅有调用 `setTransform()` 函数产生的、变换相关的更改会导致调用 `itemChange()`。Qt 4.7 将扩展到 Qt 4.6 引入的其他函数——`setRotation()`、`setScale()` 和 `setTransformOriginPoint()`)。

我们仅需要对影响场景保存/载入的更改发出 `dirty()` 信号,例如在场景文件中不记录项的选中状态,所以更改项的选中状态不会使场景变“脏”。

因为所有的自定义项都重新实现了 `itemChange()` 函数,所以我们把更改是否使场景变“脏”的判断提取出来放在全局的 `isDirtyChange()` 函数中。

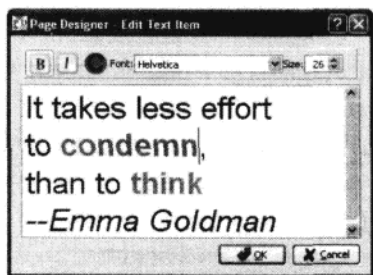


图 12.3 Page Designer 应用程序中编辑模式下的 `TextItemDialog`

```
bool isDirtyChange(QGraphicsItem::GraphicsItemChange change)
{
    return (change == QGraphicsItem::ItemPositionChange ||
            change == QGraphicsItem::ItemPositionHasChanged ||
            change == QGraphicsItem::ItemTransformChange ||
            change == QGraphicsItem::ItemTransformHasChanged);
}
```

当项的位置或变换更改时该函数返回 true;其他的更改则返回 false。

我们将在后面介绍图形项变换的单独的小节中介绍最后几个 TextItem 的成员函数, setAngle() 和 setShear()。但先看看全局的、用来把 TextItem 写入数据流中的 QDataStream 操作符。

```
QDataStream &operator<<(QDataStream &out, const TextItem &textItem)
{
    out << textItem.pos() << textItem.angle()
        << textItem.shearHorizontal() << textItem.shearVertical()
        << textItem.zValue() << textItem.toHtml();
    return out;
}
```

对于所有的自定义项的 <<() 操作符我们写出项在场景坐标系中的位置、角度、修剪和它的 z 值 (“页面编辑器”程序没有提供修改 z 值的方法, 添加这部分功能作为练习留给读者)。接着写项特定的数据, 本例中只是 HTML 格式 (以保留字体、颜色和其他格式信息) 的项的文本。

如前所述, 这个操作符不光用于把 TextItem 写入文件, 还负责在项复制或剪切到剪贴板时写入 QByteArray 对象中。

```
QDataStream &operator>>(QDataStream &in, TextItem &textItem)
{
    QPointF position;
    double angle;
    double shearHorizontal;
    double shearVertical;
    double z;
    QString html;
    in >> position >> angle >> shearHorizontal >> shearVertical >> z
        >> html;
    textItem.setPos(position);
    textItem.setAngle(angle);
    textItem.setShear(shearHorizontal, shearVertical);
    textItem.setZValue(z);
    textItem.setHtml(html);
    return in;
}
```

这个“获取操作符”(get from operator)是用来读入由“写入操作符”(put to operator)写入的数据的。因为总是和 QDataStream 配合工作, 按照写入时相同的类型和顺序读取非常重要。与“写入操作符”配合工作时, “读入操作符”用于两个目的——读入文件或从系统剪贴板保存的 QByteArray 中粘贴项到场景中。

12.4.1.1 图形项的变换

如果项有旋转 (但无修剪或缩放), 那么可以通过更改旋转为其原始角度 (如 0°) 的方式恢复到它的原始位置。与此类似, 如果项有水平修剪 (但无旋转、缩放和垂直修剪) 那么可以通过设置水平修剪回原始值 (如 0.0) 将其恢复。同样的方法适用于垂直修剪。但是如果上述变换组合起来 (旋转加修剪、垂直修剪加水平修剪或者旋转加缩放及其他情况), 恢复角度到 0°、修剪值到 0.0 以及缩放系数到 1.0 将无法恢复初始图像。事实上一般无法用简单的反操作 (如应用“相反的”变换) 撤销变换 [这是因为通常情况下 (以及在 Qt 内部) 变换用矩阵方式表示, 而并非所有的矩阵都是可逆的]。

在 Page Designer 程序中,我们解决了该问题。方法是不叠加变换(不改变原有的变换),而是每次都创建并设置新的变换,这样可以完全避免“撤销”的问题。通过在每个项中维护保存了角度和修剪值的私有成员变量可以实现,也就是说每当用户更改角度或修剪值时,我们为项创建一个新的 QTransform。

由于控制水平和垂直缩放的代码和逻辑与控制修剪的逻辑几乎一模一样,并不能教我们任何已经学到的东西以外的内容,所以这里就不提供这部分的实现,而是把添加缩放的支持作为练习留给读者。

为支持变换,要给每种变换提供一个设置相关变换分量(角度、修剪值及其他)的函数,以及一个私有的 updateTransform() 方法,用于创建和设置合适的 QTransform。我们先从 setAngle() 和 setShear() 槽看起,然后看看 updateTransform() 方法。

```
void TextItem::setAngle(double angle)
{
    if (isSelected() && !qFuzzyCompare(m_angle, angle)) {
        m_angle = angle;
        updateTransform();
    }
}
```

当项被旋转即用户更改了 TransformWidget 的角度微调框时,调用该函数。由于每个项都连接到了角度微调框的 valueChanged() 信号,我们必须小心地只把更改应用到选中的项上。另外我们把 qFuzzyCompare() 作为优化函数,确保仅当角度的变化足够产生影响时才应用变换。

```
void TextItem::setShear(double shearHorizontal, double shearVertical)
{
    if (isSelected() &&
        (!qFuzzyCompare(m_shearHorizontal, shearHorizontal) ||
         !qFuzzyCompare(m_shearVertical, shearVertical))) {
        m_shearHorizontal = shearHorizontal;
        m_shearVertical = shearVertical;
        updateTransform();
    }
}
```

如同 setAngle() 方法,仅当项选中并且至少一个新修剪值与旧值的差别足够大才修改项的修剪属性。

```
void TextItem::updateTransform()
{
    QTransform transform;
    transform.shear(m_shearHorizontal, m_shearVertical);
    transform.rotate(m_angle);
    setTransform(transform);
}
```

私有函数用于把变换应用到项上。先创建一个新的 QTransform[保存恒等矩阵(identity matrix, 即无旋转、缩放或修剪)], 然后应用修剪、旋转角度, 在最后设置的项创建变换。

注意,不需要显式地发出 dirty() 信号;由于设置了 ItemSendsGeometryChanges 标志, itemChange() 函数被调用并发出了 dirty() 信号。

前面提到了主窗口提供了“清除变换”动作。作为本小节变换部分的结尾,我们介绍一下当这个动作被触发时调用的主窗口的槽。

```
void MainWindow::editClearTransforms()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    Q_ASSERT(!items.isEmpty());
    QListIterator<QGraphicsItem*> i(items);
```

```

while (i.hasNext()) {
    if (QObject *item = dynamic_cast<QObject*>(i.next())) {
        if (item->property("angle").isValid()) {
            item->setProperty("angle", 0.0);
            item->setProperty("shearHorizontal", 0.0);
            item->setProperty("shearVertical", 0.0);
        }
    }
}
transformWidget->setAngle(0.0);
transformWidget->setShear(0.0, 0.0);
setDirty(true);
}

```

该函数通过设置选中项的旋转角度为 0.0° 及修剪为 0.0, 清除了项的变换。然后为了让 TransformWidget 正确地反映出选中项将角度设为 0.0° 、修剪设为 0.0 之后的变换状态, 我们要刷新 TransformWidget。最后调用 setDirty(), 因为变化很明显(使用 dynamic_cast <>() 要用到 RTTI, 但可以绕过该函数)。

我们用一个断言(assertion)检查列表是否为空(毕竟我们只在至少有一个选中项的时候才能使“清空变换”动作), 如果列表为空, 则表示程序有 bug。

严格来讲, 这个函数不是必需的, 因为用户可以通过选中相关的项然后在 TransformWidget 中更改它们的角度和修剪值为 0.0 达到同样的效果, 但有了这个函数更方便——用户仅需要点击一个工具栏按钮, 而不用设置三个微调框为 0.0。如果程序加入了缩放的支持, 那么这个函数就更方便了, 因为没有这个函数, 用户就需要设置 5 个微调框为 0.0。

12.4.2 增强现有的图形项

本小节将介绍从 QObject 和 QGraphicsRectItem 派生而来的 BoxItem 子类。这个类和前一小节中的 TextItem 类相比有更多的定制行为。特别是用户可以通过拖曳或通过方向键同时按下 Ctrl 键(Mac OS X 中为⌘键)移动方框项, 还可以通过 Shift 加点击项的一个角, 或者 Shift 键按下时按下方向键更改方框的大小。

```

class BoxItem : public QObject, public QGraphicsRectItem
{
    Q_OBJECT
    Q_PROPERTY(QBrush brush READ brush WRITE setBrush)
    Q_PROPERTY(QPen pen READ pen WRITE setPen)
    ...
public:
    enum {Type = BoxItemType};
    explicit BoxItem(const QRect &rect, QGraphicsScene *scene);
    int type() const { return Type; }
    ...
signals:
    void dirty();
public slots:
    void setPen(const QPen &pen);
    void setBrush(const QBrush &brush);
    ...
protected:
    QVariant itemChange(GraphicsItemChange change,
                        const QVariant &value);
    void keyPressEvent(QKeyEvent *event);
    void mousePressEvent(QGraphicsSceneMouseEvent *event);
    void mouseMoveEvent(QGraphicsSceneMouseEvent *event);
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event);
    ...
};

```



这里省略了角度和修剪属性、访问函数和槽,因为这些与上一小节看到的 `TextItem` 相同。

构造函数的实现(在此未列出)与上一小节中看到的 `TextItem` 的构造函数很相似,唯一的不同是为 `BoxItem` 设置了一个额外的标志(`QGraphicsItem::ItemIsFocusable`),这样方框项就可以接受键盘事件,当然还设置 `QGraphicsRectItem` 基类的矩形为传入构造函数的矩形。`BoxItem` 除了有与 `TextItem` 相同的 `double` 型的私有变量 `angle` 和 `shear` 外,还有一个布尔型的私有变量 `m_resizing`。

`BoxItem` 有用于设置画笔、画刷、角度和修剪值的公有槽,并且重新实现了保护型的 `itemChange()`、`keyPressEvent()`、`mousePressEvent()`、`mouseMoveEvent()` 和 `mouseReleaseEvent()` 事件处理函数。我们将介绍所有的事件处理函数,除了和 `TextItem` 相同的 `itemChange()` 函数。槽中只介绍 `setBrush()`,因为 `setPen()` 槽与之在结构上完全相同,而 `setAngle()` 和 `setShear()` 和看到过的 `TextItem` 的槽一样。首先介绍 `setBrush()` 槽。

```
void BoxItem::setBrush(const QBrush &brush_)
{
    if (isSelected() && brush_ != brush()) {
        QGraphicsRectItem::setBrush(brush_);
        emit dirty();
    }
}
```

由于所有带画刷属性的项都连接到 `BrushWidget`,我们只把画刷的更改应用到选中的项上——而且仅当新画刷与旧画刷不同时。如果设置了新的画刷,就发出 `dirty()` 信号通知所有对此感兴趣的对象。

```
void BoxItem::keyPressEvent(QKeyEvent *event)
{
    if (event->modifiers() & Qt::ShiftModifier ||
        event->modifiers() & Qt::ControlModifier) {
        bool move = event->modifiers() & Qt::ControlModifier;
        bool changed = true;
        double dx1 = 0.0;
        double dy1 = 0.0;
        double dx2 = 0.0;
        double dy2 = 0.0;
        switch (event->key()) {
            case Qt::Key_Left:
                if (move)
                    dx1 = -1.0;
                dx2 = -1.0;
                break;
            ...
            default:
                changed = false;
        }
        if (changed) {
            setRect(rect().adjusted(dx1, dy1, dx2, dy2));
            event->accept();
            emit dirty();
            return;
        }
    }
    QGraphicsRectItem::keyPressEvent(event);
}
```

我们已经为移动项(`Ctrl + 方向键`或 `Mac OS X 上⌘ + 方向键`)和更改项的大小(`Shift + 方向键`)提供了键盘支持[在 `QGraphicsView` 中使用方向键而不使用键盘修饰键(modifier)将滚动视图]。在这两种情况下都要根据按下的方向键计算 x 和 y 坐标的差值,按计算出的坐标差值调整当前矩形,获得一个新的矩形,然后用新矩形重置方框的矩形。如果处理了键盘按下,则接受该事件(这

样 Qt 就会忽略该事件), 鉴于移动或改变大小是明显的改变, 需要发出 `dirty()` 信号, 然后返回; 否则把键盘处理传递给基类去处理(本例中意味着键盘事件将被忽略)。

```
void BoxItem::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    if (event->modifiers() & Qt::ShiftModifier) {
        m_resizing = true;
        setCursor(Qt::SizeAllCursor);
    }
    else
        QGraphicsRectItem::mousePressEvent(event);
}
```

由于这里设置了 `QGraphicsItem::ItemIsMovable` 标志, 用户可以通过点击和拖曳来简单地移动方框项。我们通过提供对改变大小的支持(包括刚刚看到的使用键盘, 还有通过 `Shift + 点击然后拖曳`)扩展了方框项的行为。

如果用户在 `Shift` 键按下时点击, 通过设置 `m_resizing` 为 `true` 的同时更改鼠标光标样式来启动改变大小的动作:

```
void BoxItem::mouseMoveEvent(QGraphicsSceneMouseEvent *event)
{
    if (m_resizing) {
        QRectF rectangle = rect();
        if (event->pos().x() < rectangle.x())
            rectangle.setBottomLeft(event->pos());
        else
            rectangle.setBottomRight(event->pos());
        setRect(rectangle);
    }
    else
        QGraphicsRectItem::mouseMoveEvent(event);
}
```

如果改变大小的动作在进行中, 那么就简单地更新方框项的左下角和右上角以匹配当前鼠标的位置。而一个更复杂的算法应该分别处理 x 和 y 坐标, 但我们的实现足以阐明这个意图。

```
void BoxItem::mouseReleaseEvent(QGraphicsSceneMouseEvent *event)
{
    if (m_resizing) {
        m_resizing = false;
        setCursor(Qt::ArrowCursor);
        emit dirty();
    }
    else
        QGraphicsRectItem::mouseReleaseEvent(event);
}
```

一旦用户在改变大小的操作过程中放开了鼠标, 那么就停止改变大小的动作, 并恢复鼠标光标样式。因为改变项的大小是一个明显的更改, 还要发出 `dirty()` 信号。

目前已经讲解了 `BoxItem` 类几乎所有的内容。这里不再介绍流操作符, 因为它们和 `TextItem` 使用得非常相似, 仅有一点不同, 写入和读取的数据是方框的矩形、画笔和画刷, 而不是 HTML 字符串。

12.4.3 从头创建一个自定义图形项

在本小节将观察 `SmileyItem` 类, 即一个同时提供了实现项的外观及其行为的代码的 `QGraphicsItem` 子类。对于 Qt 4.5 及更早版本, 需要从 `QObject` 和 `QGraphicsItem` 派生, 但 Qt 4.6 及之后的版本只需要从 `QGraphicsObject` 派生即可。由于头文件中的定义相当长, 这里省略了属性及其访问函数和槽(与 `BoxItem` 完全相同)以及私有部分。

```

class SmileyItem : public QGraphicsObject
{
    ...
public:
    enum Face {Happy, Sad, Neutral};
    enum {Type = SmileyItemType};

    explicit SmileyItem(const QPoint &position,
                        QGraphicsScene *scene);
    int type() const { return Type; }
    ...
    Face face() const { return m_face; }
    bool isShowingHat() const { return m_showHat; }

    void paint(QPainter *painter,
              const QStyleOptionGraphicsItem *option, QWidget *widget);
    QRectF boundingRect() const;
    QPainterPath shape() const;

signals:
    void dirty();

public slots:
    ...
    void setFace(Face face);
    void setShowHat(bool on);
    void edit();

protected:
    QVariant itemChange(GraphicsItemChange change,
                        const QVariant &value);
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent*) { edit(); }
    void contextMenuEvent(QGraphicsSceneContextMenuEvent*) { edit(); }
    ...
};

```

除了标准的 Type 枚举类型,还提供了一个该类特定的枚举类型,用来表示该项拥有何种笑脸。另外由于我们自己处理项的外观,还须提供 paint()、boundingRect() 和 shape() 方法的重新实现。

为了提供项的行为,提供了多个公有槽(这里仅列出了几个)用于设置项的属性,还有 dirty() 信号和 edit() 槽以遵循 Page Designer 应用程序中采用的惯例。

一如往常,我们重新实现了 itemChange() 函数,这样就可以在适当的时候发出 dirty() 信号了。另外,还提供了两种调用项的 edit() 槽的方法——双击鼠标(Page Designer 程序的惯例之一)和上下文菜单调用(因为对该项来说编辑动作就是弹出一个菜单)。

私有数据(在此未列出)用于追踪画笔、画刷、笑脸、帽子是否显示以及角度和修剪值。我们还保存了两个绘制路径——笑脸的路径是个简单的椭圆形,而帽子则是一个更复杂的8个端点的多边形。

构造函数(在此未列出)与 TextItem 相似,唯一的不同是我们为私有数据设置初始值(快乐的笑脸、无画笔、黄色画刷、不显示帽子、角度为 0.0° 以及修剪值为 0.0),然后调用 createPaths() 帮助函数创建笑脸和帽子的路径。

```

const int SmileySize = 60;
...
void SmileyItem::createPaths()
{
    m_facePath.addEllipse(-SmileyHalfSize, -SmileyHalfSize,
                          SmileySize, SmileySize);

    const int LeftX = -(SmileyHalfSize + (SmileyMargin / 2));
    const int RightX = SmileyHalfSize - (SmileyMargin / 2);
    const int Y = -SmileyHalfSize + (SmileyMargin / 2);
    QPolygonF polygon;
    polygon << QPointF(LeftX * 1.4, Y + SmileyMargin)
    ...
}

```

```

        << QPointF(LeftX * 1.4, Y + SmileyMargin);
    m_hatPath.addPolygon(polygon);
}

```

我们省略了大多数用于帽子多边形的点,以及大多数的常量。帽子的形状可以在图 12.4 所示的表情符号项中看到。

```

void SmileyItem::setFace(Face face)
{
    if (m_face != face) {
        m_face = face;
        update();
        emit dirty();
    }
}

```

该函数用于改变笑脸。如果新的笑脸设置与当前设定不同,那么私有变量被更新,update() 函数被调用以安排重绘,由于改变笑脸是个明显的更改,dirty() 信号被发出。

这里不再展示 setPen()、setBrush()、setAngle()、setShear() 和 itemChange() 函数,因为所有的函数与前面两个小节看到的函数代码完全相同。

```

void SmileyItem::setShowHat(bool on)
{
    if (m_showHat != on) {
        prepareGeometryChange();
        m_showHat = on;
        emit dirty();
    }
}

```

该槽用于显示或隐藏帽子。显而易见,依赖于帽子是否显示,项的边界矩形和形状会有所不同,所以必须通知图形/视图架构项的几何位置将发生变化。可以通过调用 QGraphicsItem::prepareGeometryChange() 简单实现——它会相应调用 update() 函数,这是不需要自己调用该函数的原因。

```

void SmileyItem::edit()
{
    QMenu menu;
    QAction *showHatAction = createMenuAction(&menu, QIcon(),
        tr("Show Hat"), m_showHat);
    connect(showHatAction, SIGNAL(triggered(bool)),
        this, SLOT(setShowHat(bool)));
    menu.addSeparator();
    QActionGroup *group = new QActionGroup(this);
    createMenuAction(&menu, QIcon(":/smileysmile.png"),
        tr("Happy"), m_face == Happy, group, Happy);
    ...
    AQP::accelerateMenu(&menu);
    QAction *chosen = menu.exec(QCursor::pos());
    if (chosen && chosen != showHatAction)
        setFace(static_cast<Face>(chosen->data().toInt()));
}

```

对于这种项类型,我们设置了编辑动作作为弹出一个上下文菜单。菜单提供显示或隐藏帽子以及选择笑脸的功能,如图 12.4 所示。

我们省略了平静的面孔(neutral)和悲伤的面孔(sad face)创建过程,因为代码在结构上与创建快乐的笑脸(happy face)相同——而在三种情况下我们都把大部分工作交给帮助函数完成。我们为 showHatAction 创建一个直接的信号/槽连接,连接到项的 set-

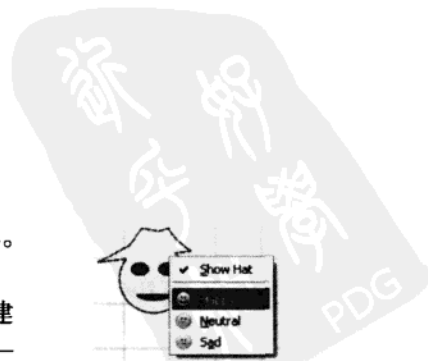


图 12.4 Page Designer 程序的表情符号上下文菜单

ShowHat()槽,这样动作就可以自行工作。但选择笑脸的动作尚未连接到任何槽,而是当用户选择一个动作时(用户还可以按下 Esc 键取消菜单,所以并不一定如此),检查用户选择的是哪个动作,如果用户选择了动作而该动作不是 showHatAction,那么必然是某个笑脸选择动作,故调用 setFace()函数。在 createMenuAction()函数里我们把对应的 Face 枚举值和每个动作关联起来,这样就能提取该值并在选择用户指定的笑脸时使用该值。

```
QAction *SmileyItem::createMenuAction(QMenu *menu, const QIcon &icon,
                                       const QString &text, bool checked, QActionGroup *group,
                                       const QVariant &data)
{
    QAction *action = menu->addAction(icon, text);
    action->setCheckable(true);
    action->setChecked(checked);
    if (group)
        group->addAction(action);
    action->setData(data);
    return action;
}
```

上面这个小而方便的帮助函数节省了 edit()槽函数中的几行代码。动作组可以确保组中任意时刻只有一个动作项被标记为复选(checked)。

现在已经介绍了提供项的行为的成员方法和槽(除了前面的小节介绍的那些之外)。流操作符与 TextItem 基本相同,只有一点不同是,写入和读取的数据是笑脸(把枚举值转换成 qint16 或从 qint16 转换得到枚举值)、画笔、画刷和帽子是否显示的标志,而不是 HTML 字符串。所以现在我们以介绍绘制的实现和 shape()及 boundingRect()函数的重新实现作为收尾。

```
void SmileyItem::paint(QPainter *painter,
                      const QStyleOptionGraphicsItem*, QWidget*)
{
    paintFace(painter);
    if (m_showHat)
        paintHat(painter);
    if (isSelected())
        paintSelectionOutline(painter);
}
```

为了仅展示上层的逻辑,不深陷于细节,该函数经过了重构。任何情况下都绘制笑脸,然后根据需要绘制帽子,最后如果项被选中就绘制表示选中的轮廓线。

```
void SmileyItem::paintFace(QPainter *painter)
{
    painter->setPen(m_pen);
    painter->setBrush(m_brush);
    painter->drawPath(m_facePath);
    int leftX = -SmileyHalfSize + SmileyMargin;
    int rightX = SmileyHalfSize - SmileyEyeWidth - SmileyMargin;
    paintEyes(painter, leftX, rightX);
    paintMouth(painter, leftX, rightX);
}
```

绘制笑脸时先绘制笑脸的路径(一个椭圆形),然后在之上绘制眼睛和嘴巴。

```
void SmileyItem::paintEyes(QPainter *painter, int leftX, int rightX)
{
    int y = -SmileyHalfSize + qRound(SmileyMargin * 1.5);
    painter->setBrush(m_brush.color().darker());
    painter->drawEllipse(leftX, y, SmileyEyeWidth, SmileyEyeHeight);
    painter->drawEllipse(rightX, y, SmileyEyeWidth, SmileyEyeHeight);
}
```

使用后台 (off-screen) 渲染

基于图形/视图的应用程序的性能取决于渲染的内容和渲染的方法。如果自定义项需要绘制复杂的形状、文字或渐变,或者应用了复杂的裁剪形状 (clipping), 那么程序的性能可能会比较糟糕。

如果缓慢而复杂的渲染是不可避免的, 一种解决方法是用图像编辑程序创建一个项的图片, 在 `QGraphicsItem::paint()` 的重新实现中载入一个 `QPixmap`, 然后绘制这个图片。多数情况下绘制单个图片的性能优于一点一点的渲染。可惜的是, 这个方案有很多局限性, 最明显的是图片没有办法显示出状态的变化 (例如当项的颜色发生改变), 以及变换——比如缩放就有可能把图片变得颗粒化 (pixelated) 和模糊不清 (fuzzy)。

另一个更通用的方案是使用 `QGraphicsItem` 提供的后台缓存 (off-screen cache)。默认情况下 `CacheMode` 为 `QGraphicsItem::NoCache`, 但如果通过调用 `QGraphicsItem::setCacheMode()` 使能缓存, 被缓存的项将渲染到图片。还可以把一个 `QSize` 对象作为第二个参数传入 `QGraphicsItem::setCacheMode()` 函数, 来手动设置图片的大小; 否则该大小将基于项的 `boundingRect()` 计算得到。在后面该项显示出来的时候, 这个图片将被重用, 完全省略对 `QGraphicsItem::paint()` 的调用。缓存的运作完全透明, 所以不论何时调用 `update()` 时, 图片都在项渲染之前更新。

我们当然可以为现有的项打开缓存功能, 比如 `QGraphicsTextItem`, 可以在缓存有意义时加快渲染的速度。但请注意, 广泛使用缓存有可能耗尽机器的显存 (graphics memory)。

缓存有两种运作模式, 决定了使用何种坐标系统。如果用 `ItemCoordinateCache` 模式, 项将以本地坐标渲染, 这个模式把项的逻辑单位转换为像素, 这意味着当缩放项或视图时, 视觉结果的质量会降低——例如放大时项会看起来颗粒化 (pixelated) 并且模糊不清。在这种模式下如果项经过变换, 缓存不变, 这使得该模式对使用动画变换 (如被旋转) 的项很理想——例如使用了很多变换的 OpenGL 程序。

第二种缓存模式——`DeviceCoordinateCache`——以设备坐标渲染项。这个模式确保完美的像素显示效果, 项发生变换则缓存重新生成以确保正确的显示结果。实际上在只移动项或者滚动视图时, 缓存一直有效, 所以缓存仅在项被旋转后才重新生成。这样, 这个模式在项被移动时很快, 远快于绘制, 而且很方便, 因为渲染效果总是很完美的。

眼睛只是用比绘制笑脸稍微深色的画刷绘制的两个椭圆。

```
void SmileyItem::paintMouth(QPainter *painter, int leftX, int rightX)
{
    int y = SmileyHalfSize - qRound(SmileyMargin * 1.1);
    int offset = 0;
    if (m_face == Neutral)
        offset = SmileyMargin;
    else {
        offset = SmileyMargin / 2;
        if (m_face == Happy)
            y -= SmileyMargin;
        else if (m_face == Sad)
            y += SmileyMargin / 2;
    }
    QPointF leftPoint(leftX + offset, y);
    QPointF rightPoint(rightX + SmileyEyeWidth - offset, y);
    QRectF mouthRect(leftPoint, rightPoint);
```




```

mouthRect.setHeight(m_face == Neutral ? SmileyMargin / 2
                    : SmileyMargin);
if (m_face == Neutral)
    painter->drawRoundedRect(mouthRect, 5, 5);
else if (m_face == Happy)
    painter->drawChord(mouthRect, 170 * 16, 200 * 16);
else
    painter->drawChord(mouthRect, 30 * 16, 120 * 16);
}

```

这是需要绘制的内容中最复杂的部分。对于平静的面孔画一个圆角矩形,但对于快乐的笑脸和悲伤的面孔我们把嘴画成一个弦状(chord),根据表情相应把末端朝上或朝下。`QPainter::drawRoundedRect()`函数接受一个矩形和用于定义圆角的椭圆半径作为参数,而`QPainter::drawChord()`函数接受一个矩形、一个起始位置的角度和一个横跨的角度作为参数——角度用 1° 的 $1/16$ 来表示。

```

void SmileyItem::paintHat(QPainter *painter)
{
    QPen pen(m_pen);
    if (pen.style() != Qt::NoPen)
        pen.setColor(pen.color().lighter());
    painter->setPen(pen);
    QBrush brush(m_brush);
    if (brush.style() != Qt::NoBrush)
        brush.setColor(brush.color().lighter());
    painter->setBrush(brush);
    painter->drawPath(m_hatPath);
}

```

如果要画帽子,就用一个颜色稍浅的画笔(或不用画笔)和一个颜色稍浅的画刷(或不用画刷),然后简单画出帽子的绘制路径。

```

void SmileyItem::paintSelectionOutline(QPainter *painter)
{
    QPen pen(Qt::DashLine);
    pen.setColor(Qt::black);
    painter->setPen(pen);
    painter->setBrush(Qt::NoBrush);
    painter->drawPath(m_showHat ? m_facePath.united(m_hatPath)
                          : m_facePath);
}

```

我们选择使用黑色的宽度为1像素的虚线装饰画笔(无缩放)、不使用画刷来绘制选中时的轮廓线,用描绘笑脸的路径或在帽子显示出来时描绘笑脸和帽子的总路径来绘制轮廓线。

```

QRectF SmileyItem::boundingRect() const
{
    QRectF rect(-SmileyHalfSize, -SmileyHalfSize, SmileySize,
                SmileySize);
    if (m_showHat)
        rect = rect.united(m_hatPath.boundingRect());
    return rect;
}

```

可以用语句 `QRectF rect(m_facePath.boundingRect())` 来得到边界矩形,但我们采用了更简单快速、基于项的大小的计算来代替。而当帽子显示出来时,扩展该矩形,并入帽子的边界矩形。

```

QPainterPath SmileyItem::shape() const
{
    QPainterPath path;
    path.addPath(m_facePath);
    if (m_showHat)

```

```
    path.addPath(m_hatPath);  
    return path;  
}
```

项的形状很好计算,因为要么是笑脸的路径,要么是笑脸的路径加上帽子的路径。

至此我们完成了对 Page Designer 应用程序的介绍,包括它的图形项。这个程序还有很多可改进之处,最明显的是添加新形状。应该很容易实现,因为每个新形状都能按照方框、表情符号或者文本项这些已经支持的模式来处理。另外去掉参考网格项并用背景画刷或通过实现 `QGraphicsView::drawBackground()` 的方式提供参考网格也是个好主意。当然在前面提到过其他一些可以添加的功能,比如分布项(distributing items)。添加对拖曳的支持也很不错,也就是允许用户从工具栏拖动一个形状(如表情符号)把它放在页面上。当然点击工具栏按钮更简单,但用拖曳的方式用户可以把项准确地放在想放的位置。另一个更大而复杂的改进是使用 Qt 的 `undo/redo`(`undo/redo`)架构,这样用户可以撤销和重做所有的动作^①。以上这些都作为练习留给读者。

现在到了 Qt 图形/视图架构介绍的尾声。我们的示例程序采用了 Qt 的经典解决方案——窗口部件的外观和行为由它自身提供。但当场景包含成千上万的项时,我们可能更倾向于使用不同的解决方案。可以依赖于 `QGraphicsScene` 和 `QGraphicsView` 提供基于项的行为,而不用每个项都做成 `QObject` 的子类。例如,可以创建一个场景或视图的子类,重新实现按键按下和鼠标事件处理函数,用不同的 `items()` 函数查看与用户交互的项,然后相应地把更改应用在项上。图形/视图架构功能丰富,而且在每个新的 Qt 发行版都有速度和质量上的改进,很值得学习和尝试,另外 Qt 提供的演示程序和示例程序学习起来也很有趣。

① Qt 提供了两个图形/视图的示例程序来说明 `undo/redo` 功能——`examples/tools/undoframework` 介绍了一些基础知识, `demos/undo` 则提供一些高级用法。Qt 的文档提供了 `undo/redo` 框架的总览,还有一篇本书作者撰写的介绍该框架的文章——using PyQt4,网址为 www.informit.com/articles/article.aspx?p=1187104。

第 13 章 动画和状态机框架

- 动画框架简介
- 状态机框架简介
- 动画和状态机的结合

Qt 4.6 引入了许多新特性,包括两个主要的框架:动画框架(animation framework,“Kinetic”项目的一部分)和状态机框架(state machine framework)。在这一章,我们将会简要介绍这两个框架^①。在 Qt 中,过去常用计时器(timer)来实现动画,当在 Qt 4.2 中引入 QTimeLine 类后,这一切变得简单了许多(之前的章节曾非常简要地介绍过这个类)。Qt 4.6 的新动画框架提供了一些更为高级、灵活且复杂的动画实现方法。

13.1 节将介绍动画框架并对第 12 章的 Page Designer 应用程序(参见示例 pagedesigner2)做一些小改动,以模拟图形项的对齐而不是让它们直接跳到指定的位置。在 13.2 节会介绍状态机框架并对第 11 章的 Petri Dish 应用程序(参见示例 petridish2)做一些小改动,使用状态机而不是全部都由我们自己来管理,以说明如何实现应用程序的逻辑关系。在 13.3 节,我们会给出一个既用到动画框架又用到状态机框架的小对话框,以说明它们是如何一起工作的,其中使用动画的是标准窗口部件而不是 13.1 节中给出的那些图形项。

13.1 动画框架简介

Qt 的动画框架相当复杂,但其基本概念则相当容易理解。其基本理论是基于 David Harel 的有限状态图(finite Statechart)理论,状态机的执行语义则使用 SCXML(State Chart XML,状态图 XML)。实际应用中,该框架建立在 QObject 和 Qt 属性系统的基础上。最简单的方法就是为要实现动画的 QObject 的每个属性都创建一个 QPropertyAnimation,并给定相应的动画持续时间值、初始值和最终值等参数。

所有对 QVariant 起作用的属性动画,及其相关属性必须是可写的[例如,它们必须都有一个设定函数(setter)]。对于 Qt 4.6,各 QVariant 类型都可用于动画,也就是说,Qt 可以对那些数据类型进行插值,包括 int, float, double, QColor, QLine, QLineF, QPoint, QPointF, QRect, QRectF, QSize 和 QSizeF。

例如,假设给定的持续时间为 5000 ms,初始和最终的几何形状由 QRect 给定。动画一旦开始,对象就会立即变成初始几何形状,然后,Qt 会用线性插值方法对中间的几何形状进行设置,并在这 5 s 内从初始几何形状变成最终的几何形状。因此,如果初始宽度是 100 像素而最终宽度是 400 像素,则 1 s 后宽度就会变成 160 像素,2 s 后变成 220 像素,3 s 后变成 280 像素,4 s 后变成 340 像素,最后,在 5 s 后就变成 400 像素(在本例中,增量 60 像素/秒是由初始宽度和最终宽度之差除以持续时间计算得到的,即 $(400 - 100)/5 = 60$ 。当然,Qt 会使用一种比秒更精细的时间粒度,所以,实际的宽度变化或许会从 100 像素变到 103 像素再变到 106 像素,等等)。

尽管动画框架很容易用于比较简单的情况,我们也能够轻松实现许多高级效果。一方面,我

^① 与之前所有章节的例子有所不同,本章中给出的例子使用与 Qt 4.6 相关的特性,将无法用 Qt 4.5 编译。

们不会受限于线性插值——Qt 提供了 `QEasingCurve` 类,它提供 40 多种图形插值算法。此外,还可对动画进行分类并通过串行或并行方式进行执行。动画框架可用于图形/视图架构(但仅限于 `QObject` 型的项),或者也可用于普通的 `QWidget`。

这一节中,我们会给出一个非常短且简单的例子,它为一些图形项添加动画;在本章的最后一节,我们会看到一个含有窗口部件的较复杂的动画。

`Page Designer` 应用程序拥有对齐动作,它允许用户对选中的两个或者更多的图形项与最左侧(或者是最右侧,或者是最顶端,或者是最低端)的项进行对齐。对齐会在瞬间内达成,各项跳动到新位置的时间是肉眼观察不到的。若能够为用户提供一些视觉反馈就好了,以说明对齐的是哪些项,到底发生了什么样的对齐等。当然,也务必小心,不要让对齐的动作太慢,因为那样会让人相当恼火。

实现对齐的代码放在 `editAlign()` 槽中,我们已在第 12 章“操作选中项”一节做过讨论。我们分三部分来回顾一下这个槽,其对齐功能是在最后一部分实现的。这里给出的是最后一部分的新版本,在循环中这次没有对每个项都调用 `QGraphicsItem::moveBy()`,而只是记录了它们所应在的新位置。

```
QList<QPointF> positions;
if (alignment == Qt::AlignLeft || alignment == Qt::AlignRight) {
    for (int i = 0; i < items.count(); ++i)
        positions << items.at(i)->pos() +
            QPointF(offset - coordinates.at(i), 0);
}
else {
    for (int i = 0; i < items.count(); ++i)
        positions << items.at(i)->pos() +
            QPointF(0, offset - coordinates.at(i));
}

animateAlignment(items, positions);
setDirty(true);
```

在执行每个循环的最后,每项都应有一个新位置。我们调用这个 `animateAlignment()` 自定义方法来执行移动,并向它传送一个项列表和一个新位置列表。

```
void MainWindow::animateAlignment(const QList<QGraphicsItem*> &items,
                                  const QList<QPointF> &positions)
{
    int duration = ((qApp->keyboardModifiers() & Qt::ShiftModifier)
                    != Qt::ShiftModifier) ? 1000 : 5000;
    for (int i = 0; i < items.count(); ++i) {
        QObject *object = dynamic_cast<QObject*>(items.at(i));
        if (!object)
            continue;
        QPropertyAnimation *animation = new QPropertyAnimation(
            object, "pos", this);
        animation->setDuration(duration);
        animation->setEasingCurve(QEasingCurve::InOutBack);
        animation->setKeyValueAt(0.0, items.at(i)->pos());
        animation->setKeyValueAt(1.0, positions.at(i));
        animation->start(QAbstractAnimation::DeleteWhenStopped);
    }
}
```

持续时间已设置成 1 s——或者在 `Shift` 键按下时,设置成 5 s。对于常规应用,1 s 就足够了,但如果希望能够看到较慢的动画效果,或者在开发时希望看到它是如何工作的,那么可以按下 `Shift` 键,用 5 s 的持续时间慢慢执行(值得注意的是, `QApplication::keyboardModifiers()` 方法会返回最后按键事件的修饰符。类似的方法还有 `QApplication::mouseButtons()`)。

我们会遍历(iterate)每一个需要对齐的项。对于每个项,都会创建一个新的 `QPropertyAnimation`, 给定需要操作的 `QObject` (还记得,除 Page Designer 应用程序中的主要网格项之外的全部图形项都是 `QObject` 对象)、需要模拟的属性的名字和一个父类[尽管有多种规避的方法,但正如在第 12 章给出的那样, `dynamic_cast < >()` 的用法仍然会让应用程序依赖于 RTTI(Run Time Information,运行时类型信息)的可用性]。

或许还记得,在回顾 Page Designer 的各个自定义项及其属性时,它们都没有 `pos` 属性。实际上,没有说明该属性是因为它与时间无关,但每个都有如下的属性声明:

```
Q_PROPERTY(QPointF pos READ pos WRITE setPos)
```

这就是我们没有看到的其他属性的声明。`QGraphicsItem` 基类提供了获取函数(getter)和设定函数(setter),因而获得 `pos` 属性就不再需要其他什么东西了。

一旦动画属性创建完成,就可以设置持续时间、动画曲线(easing curve,该项是可选项,默认为 `QEasingCurve::Linear`)和初始、最终的属性值。`QPropertyAnimation::setKeyValueAt()` 方法用来设置动画中特定点的属性值,其中的 0.0 是起点,1.0 是终点。在这里,我们已经设置了 `pos` 属性的初始值作为项的当前位置, `pos` 属性的最终值为项要对齐的位置。也可以设置中间的值,例如,在 0.5、0.3 或 0.6 等处的值。如果这里只有一个起始值和一个最终值,就应当改用 `QPropertyAnimation::setStartValue()` 和 `QPropertyAnimation::setEndValue()`,它们两个都带一个简单的 `QVariant`;但我们更倾向于使用 `setKeyValueAt()`,因为它更灵活些。

随着动画设置的完成,可调用 `QPropertyAnimation::start()` 开始执行。默认情况下,属性动画会一直留在内存中,直到其父类被删除为止,但在用完以后,我们就不再需要这些动画了,所以可以给 `start()` 方法传递一个删除策略(deletion policy),确保动画一结束就删除自己。

我们选用的动画曲线 `QEasingCurve::InOutBack` 会提供一种有趣的效果。例如,用户如果选中一些项并选择左对齐(Align Left),你所看到的是,这些项会朝右移动(错误的方向),但只移动一点点。然后,它们就会弹回左侧,但不是停在最左侧位置而是会冲出去。最后,它们会弹回来并与之前最靠左的项进行左对齐。

得益于 Qt 插值法的应用,所有这些移动都很平滑,图 13.1 是左对齐的效果示意图。

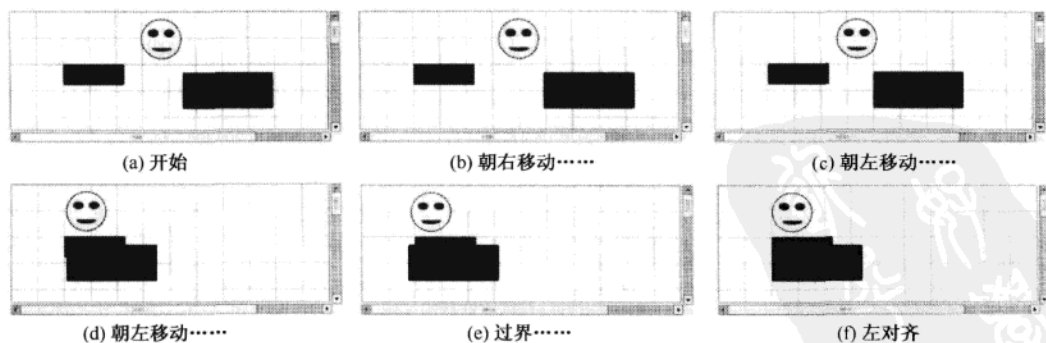


图 13.1 用 InOutBack 动画曲线进行左对齐

有一点要说明的是,这个动画是以串行方式执行的。这在实际中不是什么问题,因为 `start()` 调用几乎可以即刻返回值。然而,如果希望确保动画以并行方式出现,我们可以像下面给出的 `animateAlignment()` 实现方法那样用 `QParallelAnimationGroup` 来实现这一想法。

```
void MainWindow::animateAlignment(const QList<QGraphicsItem*> &items,
                                  const QList<QPointF> &positions)
{
    int duration = ((qApp->keyboardModifiers() & Qt::ShiftModifier)
                    != Qt::ShiftModifier) ? 1000 : 5000;
    QParallelAnimationGroup *group = new QParallelAnimationGroup;
    for (int i = 0; i < items.count(); ++i) {
        QObject *object = dynamic_cast<QObject*>(items.at(i));
        if (!object)
            continue;
        QPropertyAnimation *animation = new QPropertyAnimation(
            object, "pos", this);
        animation->setDuration(duration);
        animation->setEasingCurve(QEasingCurve::InOutBack);
        animation->setStartValue(items.at(i)->pos());
        animation->setEndValue(positions.at(i));
        group->addAnimation(animation);
    }
    group->start(QAbstractAnimation::DeleteWhenStopped);
}
```

这里,并非每个动画一创建完就启动,而是需要先把它添加到 `QParallelAnimationGroup` 中,并且仅在最后才会启动动画。`start()` 调用将会同时启动群组中的所有动画,这是由删除策略造成的,因为所有动画一旦完成,无论是群组还是群组包含的动画都会被删除掉。还要注意的,在类似于这种情况的地方使用 `setStartValue()` 和 `setEndValue()` 会更方便些,因为这里没有我们想用的中间值。

在创建特殊效果时,Qt 的动画框架易于使用且非常有效。在本章的最后一节,当我们把它与状态机结合起来,并用它模拟标准 `QWidget` 时,我们还会再次用到它。

13.2 状态机框架简介

状态机框架提供一种维护复杂应用程序状态的方法。对于简单对话框和主窗口,该框架未能提供什么真正的好处——实际上,这需要有大量代码。然而,随着复杂程度的增加,状态机框架变得越来越有吸引力,因为它有很好的弹性,从而使复杂状态的管理会比用手工管理简单得多。

对于某些类型应用程序的用户接口,比如计算器和媒体播放器,本质上是带状态的。使用变量跟踪这些状态(比如度/弧度、播放/暂停等)需要相当小心,尤其是在有嵌套状态时更是如此。在诸如以状态机图(state machine diagram)为帮助文档、以一系列经常在添加或移除间进行简单切换的状态维护等情况下,使用状态机就会非常有优势。

使用 Qt 的状态机框架相当简单,只需理解一些基本概念。这与动画框架类似,它同样非常依赖于 `QObject` 和 Qt 的属性系统。我们通过创建 `QStateMachine` 开始搭建一个状态机。然后再创建所需的一些状态(它们是 `QState` 或 `QFinalState` 的示例),并创建所有给定的三元状态组(`QObject`、属性、值)中的每一个状态,以便让状态机知道:在给定的状态中,对象的属性值一定会设置成给定的值。各个状态一旦创建完毕,我们就会创建那些切换(transition)——它们会说明一个状态是如何转变成另一个状态的。例如,单击某个特定状态下的按钮可能会让状态机切换到另一种状态下。

无论状态何时发生改变,它所做的都是发射一个 `exited()` 信号,而新进入的状态则会发射一个 `entered()` 信号。当(如果)状态机结束,它会发射一个 `finished()` 信号。

一旦建立完所有东西,就可以告诉状态机使用哪个状态作为其初始状态,然后就可以调用 `QStateMachine::start()` 来启动状态机。

就像这里所说的那样,状态机框架非常强大和灵活,但它的确与我们截至目前所提到的那些功能还相差甚远。例如,状态应可以分组,状态历史应可以追踪——以便保存或者恢复状态,应可以建立并行状态,等等。该框架应该并不只局限于图形,而应该可以很容易地用于网络通信协议的建模中。

这一节通过创建一个新版本的 Petri Dish 程序(在 petridish2 示例目录中,如图 11.3 所示)示例,来说明实际中是如何使用状态机的。我们还记得,这个应用程序有 4 个按钮,即 Start、Pause/Resume、Stop 和 Quit,一个 Initial 计数器微调框,一个 Show ID 复选框。仿真状态共有运行、暂停或停止 3 种状态,每种情况下都必须确保启用或者禁用正确的窗口部件,并且在 Pause/Resume 按钮上显示出正确的文本。在该程序的初始版本(petridish1)中,我们是通过枚举变量 SimulationStateenum 和 start()、pauseOrResume()和 stop()槽来实现这些操作的,它们不仅实现了相应的仿真行为,而且还会启用/禁用窗口部件并设置 Pause/Resume 按钮上的文字。

对于状态机 Petri Dish,就不再需要枚举变量了,在建立仿真时也只需关注 start()槽,原来 13 行的 pauseOrResume()槽会被两行的 pause()槽替换掉,原来 6 行的 stop()槽则会被完全删除。虽然如此,状态机版的 mainwindow.cpp 文件大约有 40 行,比初始文件还要长。这是因为对状态机进行了更多的设置。然而,如果状态的复杂程度或者数量有所增加,在某些时候,状态机版应当肯定会有更少的代码行,因为它的伸缩性要好得多。

在深入研究代码之前,我们最好是能够先做一个计划,以用来识别那些必要的状态和它们之间的切换。有 3 种显而易见的状态是必需的:停止、运行和暂停。但除此之外,我们将创建一个初始状态,可在那里为应用程序做一些必要的初始设置,在可用的最终状态中做一些清理和停止应用程序的操作。由于希望用户能够在任何时候都可退出应用程序,我们还需要创建一个“常规”(normal)状态,由其作为停止、暂停和运行状态的父状态,无论是哪种状态处于激活态,常规状态也都会处于激活状态,也就可用于向最终状态的切换。

初始状态的各项属性一创建,就可以切换到停止状态了。如果用户点击 Start 按钮,将切换成运行状态。在运行状态下,用户可以点击 Stop 按钮,此时就必须切换到停止状态,或者在用户点击 Pause/Resume 按钮时,必须可以切换到暂停状态。在暂停状态下,用户可以点击 Stop 按钮,在此情况下,必须切换回停止状态,或者点击 Pause/Resume 按钮并切换回运行状态。由于停止、运行和暂停状态都是常规状态的子状态,除了剩下的初始状态外,状态机就一直在常规状态下(及某些子状态,如运行状态)。如果用户点击 Quit 按钮,必须从常规状态(无论当前的子状态是什么)切换到最终状态。状态间的切换关系如图 13.2 所示。

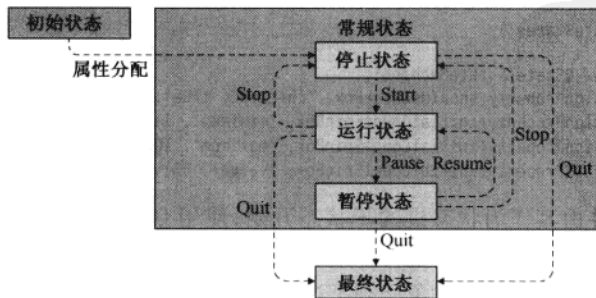


图 13.2 Petri Dish 的状态切换关系

到此为止,我们已经大致了解了要做什么事情,下面来看一下代码。先从头文件中提取代码开始。

```

    Q_PROPERTY(bool running READ running WRITE setRunning)
    ...
private:
    ...
    bool running() const { return m_running; }
    void setRunning(bool running) { m_running = running; }
    ...
    QStateMachine stateMachine;
    QState *initialState;
    QState *normalState;
    QState *stoppedState;
    QState *runningState;
    QState *pausedState;
    QFinalState *finalState;

    bool m_running;

```

利用 `running` 属性可以记录仿真是否正在运行。我们有一个状态机,每个状态都有一个 `QState`,外加一个用于最终状态的 `QFinalState`^①。还有保存 `running` 属性值的私有 `m_running` 布尔变量。

```

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), iterations(0), m_running(false)
{
    scene = new QGraphicsScene(this);
    scene->setItemIndexMethod(QGraphicsScene::NoIndex);

    createWidgets();
    createProxyWidgets();
    createLayout();
    createCentralWidget();

    createStates();
    createTransitions();
    createConnections();

    setWindowTitle(tr("%1 (State Machine)")
        .arg(QApplication::applicationName()));

    stateMachine.setInitialState(initialState);
    QTimer::singleShot(0, &stateMachine, SLOT(start()));
}

```

构造函数与原始版本的构造函数非常相似,只是这一次创建了状态和切换,建立了一些信号-槽连接(与之前相比略有不同,代码也更少),为状态机设定了初始状态,并启动了它。跟往常一样,使用一个单触发定时器来确保在开始处理进程前完全创建了窗口。现在,我们会再次回顾一下 `createStates()`、`createTransitions()` 和 `createConnections()` 这三个方法。为便于解释,我们将用 4 个简短的部分来回顾 `createStates()` 方法。

```

void MainWindow::createStates()
{
    initialState = new QState(&stateMachine);
    initialState->assignProperty(showIdsCheckBox, "checked", true);
    initialState->assignProperty(initialCountSpinBox, "minimum", 1);
    initialState->assignProperty(initialCountSpinBox, "maximum", 100);
    initialState->assignProperty(initialCountSpinBox, "value", 60);
}

```

先从创建第一个状态并由其作为状态机的子状态开始,使用 `QState::assignProperty()` 方法设置一些属性三元组。这些调用并不会把给定对象的属性设置成给定的值!相反,它们记录的内容是否进入状态、何时进入状态,而给定的对象必须把这些属性设置成特定值。

① 因为受 Windows 和 Mac OS X 上 Qt 4.6.0~4.6.2 中某个已知 bug 的影响,对于这个例子,不得不把 `QStates` 和 `QFinalState` 的成员变量替换成 `.cpp` 文件中使用 `#if` 声明的模块变量,以防止该应用程序在启动时就崩溃了。


```
normalState = new QState(&stateMachine);

runningState = new QState(normalState);
runningState->assignProperty(startButton, "enabled", false);
runningState->assignProperty(initialCountSpinBox, "enabled",
                             false);
runningState->assignProperty(stopButton, "enabled", true);
runningState->assignProperty(pauseOrResumeButton, "enabled",
                             true);
runningState->assignProperty(pauseOrResumeButton, "text",
                             tr("Pa&use"));
runningState->assignProperty(this, "running", true);
```

创建的初始状态是状态机自身的一个子状态,所以也就是常规状态。常规状态仅存在于群组状态内(运行、暂停、停止),这样,无论激活的是常规状态的哪个子状态,都可以应用切换关系。要使这一机理能够工作,就必须把常规状态当成各个子状态的父类而不是使用状态机。

对于运行状态,要确保禁用 Start 按钮和 Initial 计数器微调框,Stop 和 Pause/Resume 按钮是启用的,且后者有正确的文字。我们还会把主窗口的 running 属性设置成 true。如前所述,在调用 assignProperty() 的时候不会用到这些设置——仅在(如果)进入该状态时才会用到它们。

```
pausedState = new QState(normalState);
pausedState->assignProperty(pauseOrResumeButton, "text",
                             tr("Res&ume"));
pausedState->assignProperty(this, "running", false);
```

在暂停状态下,需要设置 Pause/Resume 按钮的文字,并把主窗口的 running 属性设置成 false。在这种状态下,应当禁用 Start 按钮,但我们不必明确设置它(尽管那样做并没有什么坏处),因为在进入运行状态时就会禁用该按钮,且暂停状态仅可以从运行状态切换过来,因而就可以知道,已经正确地禁用了 Start 按钮。

```
stoppedState = new QState(normalState);
stoppedState->assignProperty(startButton, "enabled", true);
stoppedState->assignProperty(initialCountSpinBox, "enabled",
    true);
stoppedState->assignProperty(pauseOrResumeButton, "enabled",
    false);
stoppedState->assignProperty(pauseOrResumeButton, "text",
    tr("Pause"));
stoppedState->assignProperty(stopButton, "enabled", false);
stoppedState->assignProperty(this, "running", false);

finalState = new QFinalState(&stateMachine);
}
```

在停止状态下会启用 Start 按钮和 Initial 计数器微调框,禁用 Pause/Resume 按钮(并设置它的文字)和 Stop 按钮。还会把 running 属性设置成 false。

创建的最终状态仅用于当用户退出时提供一种要切换的状态(从常规状态),因为这可以让我们不必为停止、暂停和运行状态中的每一个状态都再单独创建一个切换状态。

```
void MainWindow::createTransitions()
{
    initialState->addTransition(initialState,
        SIGNAL(propertiesAssigned()), stoppedState);
    stoppedState->addTransition(startButton, SIGNAL(clicked()),
        runningState);
    runningState->addTransition(pauseOrResumeButton,
        SIGNAL(clicked()), pausedState);
    runningState->addTransition(stopButton, SIGNAL(clicked()),
        stoppedState);
    pausedState->addTransition(pauseOrResumeButton,
        SIGNAL(clicked()), runningState);
}
```

```
    pausedState->addTransition(stopButton,
                              SIGNAL(clicked()), stoppedState);
    normalState->addTransition(quitButton, SIGNAL(clicked()),
                              finalState);
}
```

创建的第一个切换是从初始状态到停止状态。只要对初始状态的各个属性赋值,这个切换就会发生。从用户角度来看,应用程序一启动,这个切换就发生了(实际上, `QStateMachine::start()` 槽一调用,窗口的构造就结束了),所以会仅进入一次初始状态,简言之,从那时起,应用程序就处于常规状态加一个自身的子状态(停止、暂停或运行)。

对于停止状态,我们创建了一个到运行状态的切换。对于运行状态,我们创建了两个切换,一个是切换到暂停状态,另一个是切换到停止状态。与之相似的是,对于暂停状态,我们会创建到运行状态和停止状态的切换。还要创建一个从常规状态(这其中包含了它所有的子状态)到最终状态的切换。

值得注意的是,在按下 `Pause/Resume` 按钮时,会发生两个不同的切换。如果在运行状态下,按下该按钮会切换到暂停状态,而如果在暂停状态下,按下该按钮会切换到运行状态。这种把同一按钮的单击解释成两个完全不同方式的能力正是状态机框架非常强大的特性(在最初的 `Petri Dish` 应用程序中,我们不得不创建一个 `pauseOrResume()` 槽,让它连接到该按钮的 `clicked()` 信号上,并使用按钮的文字来确定用户到底是处于暂停状态还是处于恢复状态;而这里使用的状态机方法就显得要简洁和简单多了)。

```
void MainWindow::createConnections()
{
    connect(showIdsCheckBox, SIGNAL(toggled(bool)),
            this, SLOT(showIds(bool)));

    connect(runningState, SIGNAL(entered()), this, SLOT(start()));
    connect(pausedState, SIGNAL(entered()),
            this, SLOT(pause()));
    connect(&stateMachine, SIGNAL(finished()), this, SLOT(close()));
}
```

第一个连接比较常见,用来更新仿真效果,实现表格 ID 的显示或者隐藏。

其他三个连接都与状态机相关。来自运行状态的 `entered()` 信号的连接说明:除了在进入运行状态时正适当地设置各个属性外,还要调用 `start()` 槽,用以建立并启动仿真的运行。同样,当进入暂停状态时,除了设置各个属性外还要调用 `pause()` 槽(该槽仅用于让窗口显得稍微有点透明,这是一种在 Windows 上非常常见,而在 Mac OS X 上非常巧妙的效果)。当状态机结束时会激活最后一个连接(例如,在进入最终状态时);此处我们只是用它来关闭应用程序。

与这个例子中用到的功能相比,Qt 的状态机框架还有许多很好的功能。虽然如此,我们已经涉猎了框架使用 and 实际应用中较常见的那些重要概念。在下一节,我们会用到几个状态机,并且会把一个状态机与 13.1 节中讲到过的动画框架相结合,以实现一些更为复杂的效果。

13.3 动画和状态机的结合

这一节,我们会回顾 `Find Dialog` 这个典型的例子(`finddialog`),它是一个用于执行某些搜索操作的对话框。该对话框使用两个独立的状态机,一个根据用户是否输入了任意的搜索文本来启用/禁用 `Find` 按钮,另一个与动画框架一起联合工作,根据 `More` 切换按钮的状态来显示或隐藏那些额外的窗口部件。

当该窗口第一次出现时, `More` 按钮是“凸起”的(例如,没被选中),那些额外的窗口部件也都

是不可见的。如果用户单击了 More 按钮,它就会变成“凹下”的样子(例如,被选中),同时,对话框会增大尺寸,以能够容纳下更多的窗口部件。一旦该窗口变大,额外的窗口部件会从 More 按钮的下方显现出来,一开始尺寸很小,字号也很小,并且是透明的。随着朝最终位置向左、向下滑动,它们会越来越长越大,字号也会变大,并且会变得越来越不透明。随着这些额外窗口部件到达正确的位置,它们会变成正确的尺寸,拥有正确的字号大小,而且会变得完全不透明。这一过程如图 13.3 所示。如果用户再次单击 More 按钮,它会返回“凸起”的样子,那些额外的窗口部件会滑动到 More 按钮的后面,逐渐收缩、字体变小并慢慢变成透明。最后,对话框会收缩起来,以适应可见窗口部件的尺寸。

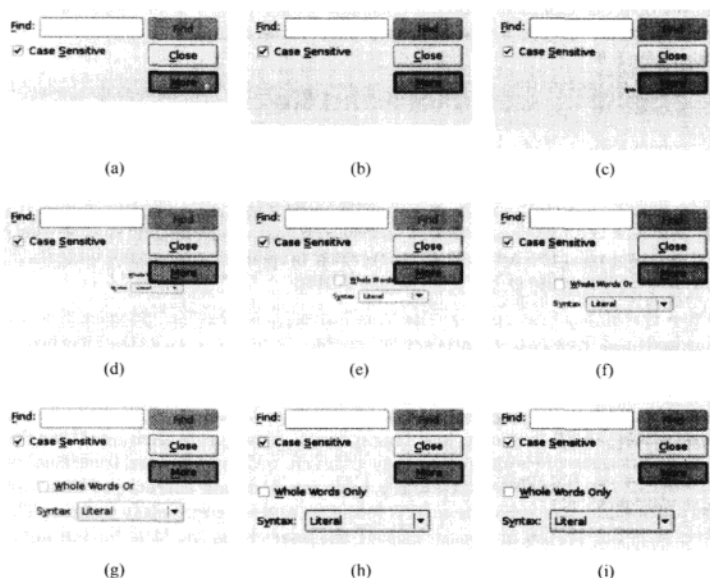


图 13.3 点击 FindDialog 的 More 按钮

先从对话框头文件中抽取的代码开始,来看一下那些私有成员变量。

```
QCheckBox *wholeWordsCheckBox;
QLabel *syntaxLabel;
QComboBox *syntaxComboBox;
QList<QWidget*> extraWidgets;

QStateMachine *findStateMachine;
QState *nothingToFindState;
QState *somethingToFindState;

QStateMachine *extraStateMachine;
QState *showExtraWidgetsState;
QState *hideExtraWidgetsState;
```

我们漏掉了那些总是可见的窗口部件(行编辑框、Find 按钮等),但却对那些根据 More 按钮的状态进行显示或者隐藏的额外窗口部件进行了说明,还保留了额外窗口部件的一个清单,以便可以一起处理它们。

两个状态机都各仅有两个状态,它们的名字应让其含义清晰。

构造函数(此处没有给出)遵循一种常见模式,创建窗口部件、布局 and 连接,然后是状态机和切换的创建。这里给出的是从 createWidgets() 方法中抽取的代码,说明了额外窗口部件的创建和配置。

```

wholeWordsCheckBox = new QCheckBox(tr("Whole Words Only"), this);
wholeWordsCheckBox->setChecked(false);

syntaxLabel = new QLabel(tr("Syntax:"), this);

syntaxComboBox = new QComboBox(this);
syntaxLabel->setBuddy(syntaxComboBox);
syntaxComboBox->addItem(tr("Literal"), QRegExp::FixedString);
syntaxComboBox->addItem(tr("Regex"), QRegExp::RegExp2);
syntaxComboBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
syntaxComboBox->setCurrentIndex(0);

extraWidgets << wholeWordsCheckBox << syntaxLabel
               << syntaxComboBox;

foreach (QWidget *widget, extraWidgets) {
    QGraphicsOpacityEffect *effect = new QGraphicsOpacityEffect;
    effect->setOpacity(1.0);
    widget->setGraphicsEffect(effect);
}

```

Qt 4.6 引入了 QGraphicsEffect 类及其子类: QGraphicsBlurEffect、QGraphicsColorizeEffect、QGraphicsDropShadowEffect 和 QGraphicsOpacityEffect。它们可在 QGraphicsScene 中与 QGraphicsItem 一起使用,或者,它们在这里还可与普通的 QWidget 一起使用。我们已经在每个额外的窗口部件上都增加了一个不透明效果,把初始透明度设置为 1.0(完全不透明)。

```

void FindDialog::createStateMachines()
{
    findStateMachine = new QStateMachine(this);
    createFindStates();
    createFindTransitions();
    findStateMachine->setInitialState(nothingToFindState);
    findStateMachine->start();

    extraStateMachine = new QStateMachine(this);
    createShowExtraWidgetsState();
    createHideExtraWidgetsState();
    createShowExtraWidgetsTransitions();
    createHideExtraWidgetsTransitions();
    extraStateMachine->setInitialState(hideExtraWidgetsState);
    extraStateMachine->start();
}

```

该方法用于创建、配置和启动这两个状态机。对额外的状态机在分割开的帮助方法(helper method)中配置了显示/隐藏状态和切换,以使每个方法都有可控制的大小。

```

void FindDialog::createFindStates()
{
    nothingToFindState = new QState(findStateMachine);
    nothingToFindState->assignProperty(findButton, "enabled", false);

    somethingToFindState = new QState(findStateMachine);
    somethingToFindState->assignProperty(findButton, "enabled", true);
}

```

查找的状态非常简单:是启用还是禁用 Find 按钮。

```

void FindDialog::createFindTransitions()
{
    nothingToFindState->addTransition(this,
        SIGNAL(findTextIsNonEmpty()), somethingToFindState);
    somethingToFindState->addTransition(this,
        SIGNAL(findTextIsEmpty()), nothingToFindState);
}

```

查找状态的切换是根据要查找的文本是空还是非空,就像查找状态一样非常简单。

我们已经可以用 updateUi() 槽实现 Find 按钮 enabled 属性的控制功能了,正如前面许多例子

中使用的那样——相较于在此使用的方法,这样做所需的代码量会更少一些。然而,使用状态机方法让我们把清晰的逻辑转换成单独的状态,并且应当会更易于维护,新增功能更简单,相比于依靠增加意大利面条式的信号-槽连接,更不容易出错。

```
void FindDialog::createConnections()
{
    connect(findTextLineEdit, SIGNAL(textEdited(const QString&)),
           this, SLOT(findTextChanged(const QString&)));
    connect(moreButton, SIGNAL(toggled(bool)),
           this, SLOT(showOrHideExtra(bool)));
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(close()));
}
```

查询文本只要改变,就应当发射一个 `findTextIsEmpty()` 或者 `findTextIsNonEmpty()` 信号。为实现这一点,需要把查找文本行编辑器的 `textEdited()` 信号与自定义的 `findTextChanged()` 槽相连接,该槽会根据查找文本的变化来发射信号。

按钮没有“选中”和“非选中”信号,而只有 `clicked(bool)` 和 `toggled(bool)`,但我们需要根据 More 按钮是凸起还是凹下发射 `showExtra()` 和 `hideExtra()` 信号。所以,我们把 More 按钮的 `toggled()` 信号连接到自定义的 `showOrHideExtra()` 槽上,该槽会根据按钮的切换状态来发射正确的信号。

第三个信号用于关闭该对话框。

```
void FindDialog::findTextChanged(const QString &text)
{
    if (text.isEmpty())
        emit findTextIsEmpty();
    else
        emit findTextIsNonEmpty();
}
```

这个槽实际上把 `QLineEdit::textEdited()` 信号转换成 `findTextIsEmpty()` 信号或 `findTextIsNonEmpty()` 信号,它们用于查找状态机在“什么也不查找”和“查找某些东西”两种状态间的切换。

```
void FindDialog::showOrHideExtra(bool on)
{
    if (on)
        emit showExtra();
    else
        emit hideExtra();
}
```

实际上,这个槽会根据 More 按钮的切换状态,把 `QPushButton::toggled()` 信号转换成 `showExtra()` 信号或 `hideExtra()` 信号。当看到额外状态机的状态切换时,我们将进一步看到是如何使用这些信号的,但首先应看看额外状态机状态的配置情况。

```
void FindDialog::createShowExtraWidgetsState()
{
    QSize size = extraSize();
    size.rheight() += minimumSizeHint().height();
    size.setWidth(qMax(size.width(), minimumSizeHint().width()));

    QList<QRectF> rects;
    int y = sizeHint().height() - margin;
    rects << QRectF(margin, y, wholeWordsCheckBox->sizeHint().width(),
                    wholeWordsCheckBox->sizeHint().height());
    y += wholeWordsCheckBox->sizeHint().height() + margin;
    int height = qMax(syntaxLabel->sizeHint().height(),
                     syntaxComboBox->sizeHint().height());
    int width = syntaxLabel->sizeHint().width();
    rects << QRectF(margin, y, width, height);
    int x = margin + syntaxLabel->sizeHint().width() + margin;
```

```

width = qMin(sizeHint().width(), size.width()) - (x + margin);
rects << QRectF(x, y, width, height);

showExtraWidgetsState = new QState(extraStateMachine);
foreach (QWidget *widget, extraWidgets) {
    showExtraWidgetsState->assignProperty(
        widget, "geometry", rects.takeFirst());
    showExtraWidgetsState->assignProperty(
        widget, "font", font());
    showExtraWidgetsState->assignProperty(
        widget->graphicsEffect(), "opacity", 1.0);
}
showExtraWidgetsState->assignProperty(this, "minimumSize", size);
}

```

对于显示的额外窗口部件的状态,我们需要为每个额外窗口部件提供一个几何结构(位置和尺寸)和一种字体,并把透明度效果设置成 1.0(完全不透明)。还需要将对话框的最小尺寸设置得足够大,以放得下那些额外窗口部件。所有这些都是在这个方法中完成的。

先从计算显示额外窗口部件所必需的宽度和高度开始,创建一个 QSize 型 size 对象,用它来说明并表示对话框应有的最小尺寸。然后创建一个 QRectF 列表,用来保存每个额外窗口部件的几何形状。上述计算一旦完成,就创建显示额外窗口部件的状态。然后,对这个状态中的每个额外窗口部件,用计算所得的相应矩形为窗口部件的几何形状属性赋值,并用对话框的字体给窗口部件的 font 属性赋值。还要把每个额外窗口部件的不透明度效果 opacity 属性设置成 1.0(一般情况下,此时不会设置任何属性;仅在处于或进入相关状态时,才会设置它们)。最后,把对话框的最小尺寸设置成之前通过计算得到的显示所有额外窗口部件所必需的尺寸值。

```

QSize FindDialog::extraSize() const
{
    const int Width = syntaxLabel->sizeHint().width() +
        syntaxComboBox->sizeHint().width() + (2 * margin);
    const int Height = wholeWordsCheckBox->sizeHint().height() +
        qMax(syntaxLabel->sizeHint().height(),
            syntaxComboBox->sizeHint().height()) + (2 * margin);
    return QSize(Width, Height);
}

```

这个帮助方法会计算必要的额外宽度值和额外高度值。额外宽度值为语法标签和语法组合框的宽度加上一些空白宽度的总和;额外高度值为整个文本检查框的宽度加上语法标签和组合框高度中最高者的高度,再外加一些空白高度的总和。

```

void FindDialog::createHideExtraWidgetsState()
{
    QRectF rect = QRectF(buttonBox->x() + (moreButton->width() / 2),
        buttonBox->height() - (moreButton->height() / 2), 1, 1);
    QFont smallFont(font());
    smallFont.setPointSizeF(1.0);

    hideExtraWidgetsState = new QState(extraStateMachine);
    foreach (QWidget *widget, extraWidgets) {
        hideExtraWidgetsState->assignProperty(
            widget, "geometry", rect);
        hideExtraWidgetsState->assignProperty(
            widget, "font", smallFont);
        hideExtraWidgetsState->assignProperty(
            widget->graphicsEffect(), "opacity", 0.0);
    }
    hideExtraWidgetsState->assignProperty(
        this, "minimumSize", minimumSizeHint());
}

```



隐藏额外窗口部件状态要比显示它简单些,因为每个额外窗口部件都拥有同样的属性值。把每个额外窗口部件的 `geometry` 属性都设置成位于 `More` 按钮中心的 1×1 的矩形, `font` 属性设置成小字体,图形效果的 `opacity` 属性设置成 `0.0`(完全透明)。最后,把对话框的最小尺寸属性设置成最小尺寸提示(这的确没有考虑那些额外窗口部件)。

```
const int Duration = 1500;

void FindDialog::createShowExtraWidgetsTransitions()
{
    QSignalTransition *transition =
        hideExtraWidgetsState->addTransition(this,
            SIGNAL(showExtra()), showExtraWidgetsState);

    createCommonTransitions(transition);

    QPropertyAnimation *animation;
    animation = new QPropertyAnimation(this, "minimumSize");
    animation->setDuration(Duration / 3);
    transition->addAnimation(animation);

    animation = new QPropertyAnimation(this, "size");
    animation->setDuration(Duration / 3);
    QSize size = extraSize();
    size = QSize(qMax(size.width(), width()),
        sizeHint().height() + size.height());
    animation->setEndValue(size);
    transition->addAnimation(animation);
}
```

创建一个从隐藏额外窗口部件状态到显示额外窗口部件状态的切换,此切换会被 `showExtra()` 信号触发。还要给该切换指定一个指针,以便可以用它添加一些动画。

我们打算用这个切换来控制 5 个属性:窗口的最小尺寸和尺寸、每个额外窗口部件的几何形状、字体和非透明效果的不透明度属性。

对于这些额外窗口部件,我们已把工作留给了帮助方法进行处理,因为同样的属性动画是用来显示和隐藏这些额外窗口部件的。

对于配置属性动画对话框的最小尺寸和尺寸属性,我们将会用处理窗口部件属性动画(例如,让它发生得更快些)所用时间的三分之一来完成。对于尺寸属性会设置一个最终尺寸值,使其足够容纳下那些额外的窗口部件。

```
void FindDialog::createCommonTransitions(
    QSignalTransition *transition)
{
    QPropertyAnimation *animation;
    foreach (QWidget *widget, extraWidgets) {
        animation = new QPropertyAnimation(widget, "geometry");
        animation->setDuration(Duration);
        transition->addAnimation(animation);
        animation = new QPropertyAnimation(widget, "font");
        animation->setDuration(Duration);
        transition->addAnimation(animation);
        if (QGraphicsOpacityEffect *effect =
            static_cast<QGraphicsOpacityEffect*>(
                widget->graphicsEffect())) {
            animation = new QPropertyAnimation(effect, "opacity");
            animation->setDuration(Duration);
            animation->setEasingCurve(QEasingCurve::OutInCirc);
            transition->addAnimation(animation);
        }
    }
}
```



无论是显示还是隐藏那些额外窗口部件都使用同样的属性动画。对于几何尺寸和字体属性(还有对于对话框属性动画)使用线性动画曲线(因为在没有明确设置曲线动画时,默认值就是线性的),但对于不透明度使用 `QEasingCurve::InOutCirc`,一开始,它会变化得比较慢,然后变化很快,再然后,在结束时,又会变化得很慢。

已配置的全部属性动画都会从其开始状态到结束状态改变属性值。然而,我们在此会使用同样的动画从隐藏额外窗口部件的状态切换到显示额外窗口部件的状态,以及从显示额外窗口部件的状态再重新切换回隐藏额外窗口部件的状态。这种方法将会奏效,因为我们已定义过这两种状态的属性值——例如,在隐藏额外窗口部件状态把不透明度效果的 `opacity` 属性设置成了 0.0;在显示额外窗口部件状态中,把 `opacity` 属性设置成了 1.0。所以,当开始动画时,根据发生的是哪种切换,`opacity` 属性值要么会从 0.0 到 1.0,要么会从 1.0 到 0.0,因而该动画会用 0.0(或者 1.0)作为起始值,用 1.0(或者 0.0)作为最终值。

```
void FindDialog::createHideExtraWidgetsTransitions()
{
    QSignalTransition *transition =
        showExtraWidgetsState->addTransition(this,
        SIGNAL(hideExtra()), hideExtraWidgetsState);

    createCommonTransitions(transition);

    QPropertyAnimation *animation = new QPropertyAnimation(this,
        "size");

    animation->setDuration(Duration);
    animation->setEndValue(sizeHint());
    transition->addAnimation(animation);
}
```

随着相反方向切换的出现,从显示额外窗口部件状态切换到隐藏额外窗口部件状态,都会使用同样的动画。此外,还要模拟对话框的尺寸属性——这一次所用的时间要与窗口部件切换的时间一样多,以便使该尺寸快速递减所用的时间不会比额外窗口部件从缩小到隐藏所花的时间多。

诸如之前讲到的那样,全部属性动画都会对 `QVariant` 进行操作,所关注的属性必须是可写的。在当前提供的动画都不能满足所需时,也有可能为那些 `QVariant` 类型提供一些自定义插补程序。一种方法是子类化 `QVariantAnimation` (`QPropertyAnimation` 的基类)并重新实现 `interpolated()` 方法。另外一种方法是实现一个带有欲插值类型起始/最终值的插值算法函数和一个 `qreal` 型 `progress` 值,返回插值后的 `QVariant` 类型值。然后,必须用 `qRegisterAnimationInterpolator <>()` 全局函数对这个函数进行重新注册。

你或许已注意到,Qt 不支持对 `QFont` 的插值,但我们却成功地对额外窗口部件的 `font` 属性进行了插值,使其从一个点插值到了默认的字体尺寸。这是通过在 `finddialog.cpp` 文件中添加一个 `fontInterpolator()` 函数并在 `FindDialog` 的构造函数中进行注册而完成的。

```
QVariant fontInterpolator(const QFont &start, const QFont &end,
    qreal progress)
{
    qreal startSize = start.pointSizeF();
    qreal endSize = end.pointSizeF();
    qreal newSize = startSize + ((endSize - startSize) *
        qBound(0.0, progress, 1.0));

    QFont font(start);
    font.setPointSizeF(newSize);
    return font;
}
```

这个插值函数返回一个字体,字体的尺寸范围是从起始字体的大小一直到最终字体的大小。`progress` 值的范围通常从 0.0 到 1.0,但对于某些动画曲线,它或许还有负值或者大于 1.0 的值,因此

必须考虑到这一点。这里仅对 `progress` 的值进行了限定,因为负值的字体大小没有意义;然而,还可以使用 `qMax(0.0, progress)` 来确保字体的大小要大于或者等于 0.0,且应该是最终字体大小的 1.0 倍以上。

插值函数要起作用的话必须重新注册。这里给出了 `FindDialog` 构造函数中使用过的一行代码:

```
qRegisterAnimationInterpolator<QFont>(fontInterpolator);
```

模板参数告诉 Qt, 给定的这个函数应当作为指定类型对象的插值函数——这里就是 `QFont` 型对象。

这样,我们就完成了 `Find Dialog` 的学习。应当说,向 `Find` 按钮添加一种效果还是比较简单的——例如,当禁用该按钮时可以打开模糊效果,而在启用按钮时则可以关闭模糊效果。额外窗口部件的移动、缩放和透明度模拟都工作得很好,但也有一个限制:这些窗口部件在切换期间会一直保留它们本来的长方形形状。如果这些窗口部件能够变形就会更具吸引力了——例如,当它们消失时,让它们的右端缩放看起来比左端缩放结束的更快一些,以使它们可以在切换过程中呈漏斗状——这是多么“瑰丽”的切换啊^①。

现在,我们已经看到了是如何用状态机框架来控制窗口部件状态的,也看到了是如何用它与动画框架一起来产生从一种状态到另一种状态的平滑视觉切换效果的。使用动画效果需要更多的代码(结果也会引入更大范围的 bug),如果与不使用动画效果相比,使用该效果会消耗更多的 CPU 周期。然而,如同现在的 GUI 应用程序通常会更注重图标(例如,菜单栏和工具栏中的那些)一样,越来越多的应用程序正在使用各种各样的动画效果。当然,动画也有许多非常具体的优点:可以更清晰地向用户说明正在发生的事情(例如,就像前一节中,在模拟各个图形项的对齐效果时),反之,用户甚至根本不会注意到这一迅速变化,或许还会试图重复这一操作,但却没有意识到这个操作早就已经完成了。



^① 有关“瑰丽”切换的更多信息请参阅 labs.qt.nokia.com/blogs/2008/12/15/genie-fx。

结 束 语

尽管这本书主要关注的是 Qt 中的重要功能领域,但还是介绍了许多 Qt 编程中的方法和技术。本书的全部例子,包括 .hpp 和 .cpp 代码文件、.qrc 资源文件以及 Qt 的 .pro 工程文件,都是使用普通文本编辑器创建和编辑的,都没有用到 Qt 设计师(Qt Designer)。如今,Qt 的开发工作并不需要这么严格。对于那些喜欢可视化地设计自己窗口的人来说,可以使用 Qt 设计师这个工具,而对于那些喜欢完整集成开发环境(Integrated Development Environment, IDE)的人来说,可以使用 Qt 创建器(Qt Creator)工具,其中集成了 Qt 设计师。要得到这些库和简单工具,只需获取一份标准 Qt 版本即可;得到 Qt SDK 版本,就可以拥有全部东西——包括 Qt 创建器。

Qt 具有众多功能,尽管如此,还有一些部件(component)可以通过其他渠道获取。一些部件可从 Qt 开发框架组(Qt Development Frameworks)获得——例如,Qt 解决方案(Qt Solutions),现在很多部件都是基于 LGPL 授权的。这些部件都提供了许多可选的窗口部件和各式各样的工具类,具体请参阅 qt.nokia.com/products/appdev/add-on-products/catalog/4。另外,还有许多第三方部件提供商。Qwt 库(qwt.sourceforge.net)提供了一些专门用于科学和工程应用方面的窗口部件和工具类。LibQxt 库(www.libqxt.org)提供了许多公用程序模块和类,包括绑定到 Berkeley DB 的库和许多种额外部件。qt-apps.org 网站为第三方的 Qt 附件(add-on)提供了一个数据仓库和大量的部件和窗口部件集。

要进一步学习 Qt,当然,阅读那些能够得到的不同类型的 Qt 书籍还是比较值得的——Qt 开发框架组在保存了一份这些书的清单。此外,ICS 公司(Integrated Computer Solutions, Inc., www.ics.com/learning/icsnetwork)常常会提供一些免费的在线视频剪辑,介绍 Qt 技术并提供一些 Qt 新发行版预览。利用要点和感兴趣技术活动的搜索,可在线观看那些 Qt 开发框架组在每年的 Qt 开发人员活动日(Qt Developer Days)上探讨的绝大多数话题(参见 qt.nokia.com/developer/learning/online/talks)。另外一个有用的 Qt 信息来源就是 Qt 季刊(Qt Quarterly),这是一份免费的在线杂志,它提供一些有关 Qt 编程的短小、及时性文章(参见 qt.nokia.com/doc/qj/index.html),尽管公司的 Qt 博客(blog.qt.nokia.com)会每分钟更新并提供多得多的信息,但比这更好的,恐怕就是 Qt 开发人员的 Qt 实验室日志(Qt Labs blog, labs.qt.nokia.com/blogs)。可以在 Qt 兴趣邮件列表上提问(这是一个点击率非常高的清单,但的确是有不少杰出的发帖人),只是一定要先用 Google 和文档编辑器检查一下,以免引起不必要的争论。

Qt 开发框架组提供了一份描述 Qt 发展方向的路线图(参见 qt.nokia.com/developer/qt-roadmap)。正如人们预料的那样,作为 Nokia 拥有的下属子公司,该路线图包含了一些支持触摸屏的新的 API,也包括一些与移动电话相关的用于信息和移动服务的新的 API。还有许多桌面开发人员同样感兴趣的东西。

在 Qt 4.7 发行版中,最大的新特性就是 Qt Quick(Qt User Interface Creation Kit, Qt 用户接口创建工具包)。它将引入一套创建用户接口的全新方法。Qt Quick 使用基于 JavaScript 的声明型语言,QML——Qt 元对象语言(Qt Meta-Object Language),它充分地使用动画和状态机框架,提供了非常灵巧且易于使用的用户接口。与传统的窗口部件和布局方法相比较,Qt Quick 的用法相当灵活,使得它很容易把生动的变换应用到各个窗口部件中。Qt Quick 非常适用于应用程序在不同设备和不同形式因素下需要呈现不同用户接口的情形。传统的 QWidget 方法非常适用于仅需要单

一用户接口的情形,这些接口设计可用于所有目标上,并可与本地平台的外观极好地集成起来。

在一段相当长的时期内,图形领域中还有大量的工作要做。相关人员正在尽最大努力提供一种简单且更类似于 Qt 的三维 API,以此作为 OpenGL API(将对其保留完整的可访问性)之上的一个层,以便尽可能大地降低 OpenGL API 的复杂性和平台相关性。

这里列举的一些东西并没有出现在路线图上,但或许它们应该列入路线图内。undo/redo(撤销/恢复)框架就应当完全集成到模型/视图(model/view)架构中。目前,要为模型提供 undo/redo 还并不是一件简单的事情,即使实现了,对于某些特殊操作,使用我们自己的方法时也应非常谨慎,而不是像常见的方法那样来确保 undo/redo 的工作正常。另一件应当做到的事情就是改善对数据库的支持。当前,还没有窗口部件能处理 NULL 值,但数据库则很希望能够支持这个值,而不是仅将其简单压缩,以用于模型/视图架构。另外,数据库的行为,尤其是有关 SQLite 的行为,好像在各个不同版本间都有些巧妙的变化。希望未来的 Qt 版本能够在这方面有所改善。令人值得欣慰的是,Item Views NG(Next Generation)项目好像正在稳步推进中,并且,与当前的模型/视图架构相比较,它将变得更强大、更灵活,也更易于使用——当你读到这句话时,或许它已准备就绪了。

还有一些“不切实际的”想法(一家之言)希望未来的 Qt 能够添加 PDF API,以支持 PDF 文件的读、写和编辑,并且应该涵盖 PDF 的全部特征。如果 Qt 能够为开放文档格式(Open Document Format,ODF)提供相似的 API,并在这种格式中改进此类文件的编写过程,那也将是非常好的事情。对最常用的压缩文档格式的读写支持将会非常有用——尤其是 .tar 文件(包括那些用 gzip 或 bzip2 压缩的文件)和 .zip 文件(对于这种文件 Qt 已经有了一些内部 API,至少可用于输出)。也希望并很高兴能够看到 Qt 对更大环境的支持——例如,用高级 API 支持终端-服务器编程。当然,也欢迎有更多的窗口部件,特别是二维和三维图形窗口部件,它们会让外围的科学计算人员和工程领域用户觉得 Qt 更为方便。至于 Qt 5,希望能看到 Qt 弃用元对象编译器(Meta-Object Compiler)。Boost 库已经表明完全有可能实现信号和槽机制,也完全有可能实现一套使用标准 C++ 的新系统,但是否有可能实现 Qt 的全部对象模型仍是一个悬而未决的问题。

当然,再没有必要等待 Qt 开发人员增加那些我们需要的特性。Qt 现在的开发比以往任何时候都更开放,因此,如果想增加一个新特性,你可以自己实现它并试着将其合并到官方的 Qt 版本中,具体情况请参阅 qt.gitorious.org。

Qt 是一个了不起的软件开发框架,Nokia 投入了巨大的精力来改进和扩展 Qt 所提供的那些功能。Qt 可用于非 GUI 编程,包括服务器、网络后台(backend)和命令行工具;Qt 也可用于 GUI 编程,支持复杂的、有吸引力的和带有高级动态用户接口的那些应用程序。Qt 可用于嵌入式设备——从面包烘烤器到移动电话再到 PDA,直到桌面系统及其以外的各种产品。

Qt 的庞大让人感到畏惧,但一旦学会了那些基础知识,Qt API 的一致性会让你在学习所需的任何其他类和模块时都变得简单易懂。出色的文档、Qt 示例和演示程序的源代码、诸如本书这样的书籍、之前提到的那些在线资源、培训课程等,当然,还有 Qt 自身的源代码,对于需要满足任何人的学习需求来说,都显得绰绰有余。与那些曾经提到过的和平台相关的库不同,Qt 让编程趣味横生,它允许我们可以在选定的平台上进行开发,而在用户所喜欢的平台上进行部署。

精选书目

The Art of Multiprocessor Programming

Maurice Herlihy, Nir Shavit (Morgan Kaufmann, 2008, ISBN 978-0123705914)

该书对多线程编程进行了完整介绍,包括许多小巧但却完全实用的例子(用 Java 语言),它们用来示范全部关键技术。最后一章对事务型内存进行了简单介绍,目的之一就是希望能够为多线程编程提供一种更高级的方法,以期不使传统技术所需的琐碎细节加重编程人员的负担。

C++ GUI Programming with Qt 4(第2版)

Jasmin Blanchette, Mark Summerfield (Prentice Hall, 2008, ISBN 0132354160)^①

该书是最理想的介绍 C++/Qt 编程的书籍,也是对 Qt 文献的完美补充。这本书讲授 C++ 程序员如何尽可能充分地使用 Qt 的各个类和模块来创建完整的应用程序。该书是 Qt 的官方教材。

C++ in a Nutshell

Ray Lischner (O'Reilly, 2003, ISBN 059600298X)

这是一本非常有用的书籍,为 C++ 语言及其标准库(包括部分作为 C++ 标准的一些 C 库)提供了可靠、简明且全面的参考。

The C++ Programming Language(第3版)

Bjarne Stroustrup (Addison-Wesley, 2000, ISBN 0201889544)

这是由 C++ 创作者撰写的标准 C++ 课本。可以认为它是一本非常有用的参考书。

Clean Code

Robert C. Martin (Prentice Hall, 2009, ISBN 0132350882)

这本书阐述了编程中很多具有“战术性”的问题,诸如良好的命名、函数设计、重构等类似话题。本书给出了很多有趣的、有用的思想,它们应当会对任何程序员都很有帮助,可改善他们的编程风格,使其程序更具可维护性(该书的例子用的是 Java)。

Code Complete: A Practical Handbook of Software Construction(第2版)

Steve McConnell (Microsoft Press, 2004, ISBN 0735619670)

这本书说明了如何构建可靠的软件,越过语言的条条框框来说明编程思想、原理和实践的内容。该书充满各种思想,将启发所有编程人员对自己编程方法的深入思考。

Design Patterns

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1998, ISBN 0201633612)

当前时期内最有影响的编程书籍,尽管阅读起来并非易事,但设计模式令人着迷,在日常程序设计工作中也具有很强的工程应用价值。

^① 本书的中文译名是《C++ GUI Qt 4 编程》(第2版),电子工业出版社已于2008年8月出版发行,详情请参阅 <http://www.phei.com.cn/bookshop/bookinfo.asp?bookcode=TP070380%20&booktype=main> 或 www.china-pub.com/42122——编者注。

Domain-Driven Design

Eric Evans (Addison-Wesley, 2004, ISBN 0321125215)

一本关于软件设计的有趣书籍,对多人参与的大型工程项目特别有用。本质上,这是一本关于创建和优化域模型(domain model,用来说明系统的设计目的)的书籍,也是通过那些包含在系统内的所有东西(并不仅仅是软件工程)来创建一种普适语言,以便与他们的思想进行交流。

Effective C++ (第3版)

Scott Meyers (Addison-Wesley, 2005, ISBN 0321334876)

对于所有 C++ 编程人员来说,这本书都很有必要阅读。它介绍了许多微妙的陷阱,给出了许多良好习惯的说明。

GUI Bloopers 2.0

Jeff Johnson (Morgan Kaufmann, 2008, ISBN 9780123706430)

别为这个古怪的名字分心,这是一本每名图形用户界面(GUI)编程人员都应认真阅读的图书。不必同意那每一条简单的建议,但在阅读本书后,对于用户接口设计,你会考虑得更为仔细和认真,也会更有洞察力。

JavaScript: The Definitive Guide (第5版)

David Flanagan (O'Reilly, 2006, ISBN 9780596101992)

对于 JavaScript/ECMA 来说,这是一本理想的参考指南,也当然是 Qt 文献中推荐用于学习 Qt-Script 的书籍。对于那些将要在 Qt 4.7 中引入 Qt Quick QML 语言的人,它也是非常有用的。

The Little Manual of API Design

Jasmin Blanchette (Trolltech/Nokia, 2009)

这篇非常短的文献(www4.in.tum.de/~blanchet/api-design.pdf)提供了一些思想,洞察了 API 设计,并从 Qt 中为其列举了绝大部分的例子。

Mastering Regular Expressions (第3版)

Jeffrey E. F. Friedl (O'Reilly, 2006, ISBN 0596528124)

这是关于正则表达式(regular expression)的标准教材,是一本非常有趣和有用的书籍。

Rapid GUI Programming with Python and Qt

Mark Summerfield (Prentice Hall, 2007, ISBN 0132354187)

这本书讲授了 PyQt4 编程,有可能是介绍 Qt 简单编程最容易的捷径。除了在书中介绍 PyQt 应用程序的编写外,PyQt 也可作为 C++/Qt 编程中相当有用的原型工具。

